SocketTools 12
Library Edition

Developer's Guide and Technical Reference
Version 12.0.3024.1286

Copyright © 2026 Catalyst Development Corporation. All rights reserved.
SocketTools™ and SocketWrench™ are trademarks of Catalyst Development Corporation.

Introduction

The SocketTools Library Edition includes standard Windows dynamic link libraries (DLLs) which can be used in a wide variety of programming languages such as Visual C++, Visual C#, Visual Basic and Delphi. The Library Edition is ideal for the developer who requires the high performance, minimum resource utilization and flexibility of a lower level interface, without the inherent overhead of ActiveX components or the .NET Framework. The SocketTools Library Edition API has over 950 functions which can be used to develop applications that meet a wide range of needs. SocketTools covers it all, including uploading and downloading files, sending and retrieving email, remote command execution, terminal emulation, and much more.

The SocketTools Library Edition includes support for the industry standard Transport Security Layer (TLS) and Secure Shell (SSH) protocols which are used to ensure that data exchanged between the local system and a server is secure and encrypted. The Library Edition implements secure versions of standard protocols such as HTTPS, FTPS, SFTP, SMTPS, POP3S, IMAPS and more. Your data is protected with TLS 1.2 or later using 256-bit encryption and full support for client certificates.

SocketTools also includes an FTP and HTTP server API, as well as a general purpose TCP server API which can be used to create secure custom server applications. There's no need for you to understand the details of certificate management, data encryption or how the security protocols work. All it takes is a few lines of code to enable the security features, and SocketTools handles the rest.

The following are some of the features in the SocketTools 12 Library Edition:

- Support for Windows 11, Windows Server 2025 and Visual Studio 2026
- Standard Windows dynamic link libraries (DLLs) with no external dependencies
- A comprehensive API with more than 20 libraries and over 950 functions
- Support for both synchronous and asynchronous network connections
- Includes libraries that can be used to create custom client and server applications
- Provides cloud-based application storage and geographical IP location services
- Support for the TLS 1.2 and TLS 1.3 protocols with 256-bit AES encryption
- Support for both implicit and explicit TLS connections
- Support for the SSH protocol and integrated support for SFTP as part of the FTP API
- Support for standard and secure proxy servers using FTP and HTTP
- Support for using client and server certificates in PKCS #12 format
- Thread-safe implementation with full support for multithreaded applications
- An extensive Developer's Guide and online Technical Reference
- Easy redistribution for any number of applications and end users

Developer's Guide

To help you get started using SocketTools, the new Developer's Guide covers a variety of programming topics related to SocketTools, as well an overview of each of the libraries included in the product. Even if you have experience working with previous versions of SocketTools, we recommend you review the Developer's Guide. If you are using a language other than Visual C++, you'll also find some very helpful information about how to make the most of SocketTools in other programming languages such as C#, Visual Basic and Delphi.

Technical Reference

The Technical Reference provides extensive documentation on all of the functions in each of the SocketTools libraries. It's here that you'll find information on how a function should be called, what the arguments are and what options are available. If it is your first time using a particular library, we recommend that you first read the overview of that library in the Developer's Guide.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Licensing Information

The SocketTools Library Edition License Agreement provides you with a single developer license and the right to redistribute the dynamic link libraries (DLLs) included with this product without any additional royalties or runtime licensing fees.

Evaluation Licenses

When you install SocketTools, you are given the option of entering a serial number or proceeding with the installation without a serial number. If you install SocketTools without a serial number, an evaluation development license will be created which is valid for a period of thirty (30) days from the date of installation. The product is fully functional during this evaluation period; however the SocketTools libraries may not be redistributed to third-parties. After the evaluation period has ended, you must either purchase a development license or remove SocketTools from your computer system.

Runtime Licensing

When you install SocketTools with a serial number, a runtime license key will be automatically generated for you and stored in a file named csrtkey12.h in the Include folder where you've installed the product. There are similarly named files for other languages, such as csrtkey12.bas for Visual Basic and csrtkey12.pas for Delphi. These files define the SocketTools runtime licensing key which must be passed to the Initialize function in the library that you are using. If you are using a language that does not have a license key already defined for it, you can create a text file that contains the license key using the License Manager utility. More information about that utility is provided below.

The runtime license key is a null-terminated string that is unique to your licensed copy of SocketTools. The runtime license key is not the same as your serial number and should only be embedded in your compiled application. If you provide source code for your product, you cannot include the runtime key with the source code. The same runtime license key should be used for all of the libraries.

If you install SocketTools with an evaluation license, then the runtime license key will be defined as a null pointer or empty string. This will allow the libraries to function on a system with a valid evaluation license, but they will not function on any other system. You must purchase a license and generate a runtime license key before redistributing an application which uses one or more of the SocketTools libraries.

License Manager

Included with your copy of SocketTools is a License Manager utility. This program enables you to see what components have been installed and registered on your system, as well as display information about your SocketTools license. If you need to create a new runtime license key, you can use this utility to do so. Select **License** | **Header File** from the menu and choose the type of file that you wish to create. For more information about how the License Manager can be used, please refer to the online help file that is included with the utility.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools 12 Upgrade Information

This section will help you upgrade an application written using a previous version of the SocketTools Library Edition. In most cases, the modifications required will be minimal and may only require a few edits and recompiling the program. However, it is recommended that you review this entire document so that you understand what changes were made and how those changes can be implemented in your software.

Supported Platforms

SocketTools 12 is supported on Windows 7, Windows Server 2008 R2 and later versions. Earlier versions of the operating system, including Windows XP and Windows Vista are no longer supported by Microsoft and cannot be used with SocketTools. We recommend using the lasted release of either Windows 10 or Windows 11.

Developers who are redistributing applications which target Windows 11 or Windows Server 2025 should upgrade to ensure compatibility with the platform and current development tools. Secure connections require TLS 1.2 or later and most services will no longer accept connections from a client using SSL 3.0 or TLS 1.0.

Development Tools

As standard Windows dynamic link libraries, SocketTools 12 may be used with virtually any programming language which can call exported functions in a DLL, either by name or by ordinal. Import libraries are provided for Visual C++ in both x86 and x64 COFF format, and for Borland's C++ compilers in OMF32 and ELF64 format. Other languages should use the conventions appropriate for calling an exported function, such as the Declare statement in Visual Basic. Although the libraries may be used with .NET languages, it is recommended you use the SocketTools .NET Edition if you are creating applications for the .NET Framework.

SocketTools Header Files

In SocketTools 12 all of the library constants, functions and C++ classes are in a single header file named **cstools12.h**. This header file also includes the error codes that are defined in a separate header file named **cserror12.h**. There are three special macros that can be defined to control how applications are built using the SocketTools libraries:

CSTOOLS_NO_LIBRARIES
Visual C++ and Borland C++ supports a pragma which can be used to automatically specify the names of libraries that the compiler should attempt to link with in order to resolve function calls. By default, it is not necessary to explicitly specify the names of the SocketTools import libraries to link to. However, if you wish to explicitly link to specific import libraries, then define this macro prior to including the cstools12.h header files.

CSTOOLS_NO_NAMESPACE
When compiling a C++ application, the SocketTools functions are defined in a namespace called SocketTools. This prevents the possibility of conflicts with functions of the same name that may be used in other libraries. If you prefer the functions to be defined in the global namespace instead, then define this macro prior to including the cstools12.h header file.

CSTOOLS_NO_CLASSES
When compiling a C++ application, class wrappers for the SocketTools API are automatically included by default. Defining this macro prevents this, and only the API function prototypes will be declared. It is important to note that this only affects programs written using C++. The class wrappers will not be included for standard C programs, regardless if this macro is defined or not.

SocketTools C++ Class Wrappers

The C++ class wrappers for the SocketTools libraries have been moved from a separate file into the cstools12.h header file, and are now automatically included whenever a C++ program is compiled. The classes have been designed for compatibility with a variety of C++ compilers; however, there are certain features

which are only available if the application is compiled using the Microsoft Foundation Classes (MFC) or Active Template Library (ATL).

For example, if an application is built using MFC, each class is derived from CObject and there are additional overloaded methods implemented which support the use of objects like CString. If the application is built without using MFC, or a different C++ compiler is used, those methods will not be available.

Upgrading Projects

If you are upgrading from earlier versions, most applications will be source code compatible with the SocketTools 12 API. In most cases, all you will need to do is install the current version, update the header file and import library references, and then recompile your application. Note that the library file names have changed, as have the export function ordinals. If your build projects use the **SocketTools11** environment variable to specify the location of the header files and import libraries, you need to change it to use **SocketTools12**. This environment variable is automatically defined during the installation process.

The SocketTools 12 libraries are not binary compatible with the earlier versions of the SocketTools libraries and cannot be used as drop-in replacements. If you have declared functions by ordinal in your application (something not typically done), those values have changed and must be updated. If you have declared the exported functions by name, you will be able to reference the new libraries using the same names.

Your runtime license key has changed for SocketTools 12, which will require you to define the new key in your application when calling the initialization functions for each API. As with previous versions of SocketTools, you can use the License Manager utility to generate a file which contains the runtime key you should use. The SocketTools 11 and earlier runtime license keys are not valid for the SocketTools 12 libraries and an error will be returned if an invalid runtime key is specified.

With SocketTools 12, secure connections will use TLS 1.2 or later. By default, the libraries will not support connections to servers which use older, less secure versions of TLS or any version of SSL. They will also no longer use weaker cipher suites that incorporate insecure algorithms, such as RC4 or MD5. For applications that require secure connections, it is recommended you use the current build of Windows 10 or Windows 11 with all security updates applied.

It is possible to force the APIs to use earlier versions of TLS for backwards compatibility with older servers. This is done by explicitly setting the ***dwProtocol*** member of the SECURITYCREDENTIALS structure to specify the protocol version required. However, this is not generally recommended because using SSL 3.0 or TLS 1.0 may cause servers to immediately reject the connection attempt.

The following lists the changes that developers should be aware of when migrating from earlier versions:

- The name of the primary header file is **cstools12.h** and the **cserror12.h** header file defines the error code values. The **csrtkey12.h** header file defines the runtime license key for this version of SocketTools. The runtime license key is normally generated automatically when SocketTools is installed with a valid serial number. It can also be generated using the License Manager utility installed with SocketTools.
- When using C or C++, the **cstools12.h** header file should be included before **Windows.h** to ensure that the compiler does not try to include the deprecated Windows Sockets 1.1 header file. Failure to do so can result in compiler warnings about duplicated macros and function prototypes.
- If you are using the Embarcadero C++ compiler, additional checks are imposed to ensure the SocketTools header files are included before **WinInet.h** and **Vcl.h** files. This prevents conflicts between our APIs and function prototypes defined in those header files.
- The name for each of the DLLs and import libraries has been changed with the new version. Applications written using previous versions of SocketTools must be updated to use the new file names.
- Some constant values have changed and added. This will not impact applications that used the defined

macros, but may impact any application which uses hard-coded numeric values, rather than constants or macros.

- Previous versions of the initialization functions, such as **HttpInitialize**, accepted a pointer to an **INITDATA** structure as their second parameter. This structure is no longer defined, and the parameter is now maintained only for backward compatibility. It has been redefined as a reserved **LPVOID** parameter and should always be **NULL**. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Functions in earlier versions of SocketTools that accepted an IPv4 address as a 32-bit integer value have changed to use the **INTERNET_ADDRESS** structure defined in the SocketTools header file. If your application stores an IP address in a binary format, you should update the code to support IPv6 addresses. It is generally recommended that you store IP addresses in their string format, and you should allocate at least 40 characters for the string. This ensures it is large enough to handle both IPv4 and IPv6 addresses.

Most of the networking APIs have an option to force the library to establish an IPv6 network connection. By default, the libraries will still give preference to using IPv4 for backwards compatibility. Note that using options which only establish connections using IPv6 may prevent applications from working correctly on older versions of Windows.

Applications which continue to use SocketTools 11 and earlier libraries can be installed side-by-side with the SocketTools 12 libraries. It is recommended that you redistribute the libraries in the same folder as your application executable, rather than a shared system folder such as C:\Windows\SysWOW64 or C:\Windows\System32. It is not recommended that you attempt to use different versions of the libraries within the same application.

Library File Names

The file names of the dynamic link libraries and import libraries have changed with the new version. The following table lists the new names. For more information, refer to the Redistribution section.

File Name	Import Library	Description
csdnsv12.dll	csdnsv12.lib	Domain Name Service Library
csftpv12.dll	csftpv12.lib	File Transfer Protocol Library
csftsv12.dll	csftsv12.lib	File Transfer Server Library
cshtmv12.dll	cshtmv12.lib	WebView Browser Library
cshtpv12.dll	cshtpv12.lib	Hypertext Transfer Protocol Library
cshtsv12.dll	cshtsv12.lib	Hypertext Transfer Server Library
csicmv12.dll	csicmv12.lib	Internet Control Message Protocol Library
csllmv12.dll	csllmv12.lib	Large Language Model Client Library
csmapv12.dll	csmapv12.lib	Internet Message Access Protocol Library
csmsgv12.dll	csmsgv12.lib	Mail Message Library
csmtpv12.dll	csmtpv12.lib	Simple Mail Transfer Protocol Library
csncdv12.dll	csncdv12.lib	File Encoding Library
csnvtv12.dll	csnvtv12.lib	Terminal Emulation Library
csnwsv12.dll	csnwsv12.lib	Network News Transfer Protocol Library

cspopv12.dll	cspopv12.lib	Post Office Protocol Library
csrshv12.dll	csrshv12.lib	Remote Command Protocol Library
csrssv12.dll	csrssv12.lib	Syndicated News Feed Library
cstimv12.dll	cstimv12.lib	Network Time Protocol Library
cstntv12.dll	cstntv12.lib	Telnet Protocol Library
cstshv12.dll	cstshv12.lib	Secure Shell Protocol Library
cstxtv12.dll	cstxtv12.lib	Text Messaging Library
cswebv12.dll	cswebv12.lib	Web Services Library
cswhov12.dll	cswhov12.lib	Whois Protocol Library
cswskv12.dll	cswskv12.lib	Windows Sockets (SocketWrench) Library

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Product Evaluation

If you install SocketTools without registering a serial number, the product will be installed with an evaluation license that is valid for a period of thirty (30) days. During this trial period, the SocketTools libraries are fully functional and can be used on the development system where the product was installed. If you need to extend the evaluation period, please contact the Catalyst Development sales office by email at sales@sockettools.com or by telephone at +1 760-228-9653, Monday through Friday during normal business hours.

Redistribution Restrictions

When using an evaluation copy of SocketTools, you cannot redistribute the libraries to another system. If you build an application using an evaluation license, it will function correctly on the development system but will fail with an error on any system that does not have a license. Once you have purchased a development license, you should recompile your application before redistributing it to an end-user. If you need to test your application on another system during the evaluation period, you must install an evaluation copy of SocketTools on that system.

Runtime Licensing

When you purchase a development license, a runtime license key will be generated for you which will be included in your applications. This runtime key must be passed to the library's Initialize function in order to use the product. If SocketTools is installed as an evaluation copy, that runtime license key will be defined as a NULL pointer or an empty string and the application cannot be redistributed until a valid key is created. If you have previously installed an evaluation copy of SocketTools and then purchased a license, you can create the runtime license key using the License Manager utility.

For more information, refer to the Licensing and Library Initialization sections.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Library Redistribution

The SocketTools license permits the use of the libraries to build application software and redistribute that software to end-users. There are no restrictions on the number of products in which the libraries may be used. However, if SocketTools has been installed with an evaluation license, any products built using its libraries cannot be redistributed to another system until a licensed copy of the toolkit has been purchased.

System Requirements

SocketTools is supported on Windows 32-bit and 64-bit desktop and server platforms. The minimum required desktop platform is Windows 7 with Service Pack 1 (SP1) installed. The minimum required server platform is Windows Server 2008 R2 with Service Pack 1 (SP1) installed. It is recommended that the current service pack be installed for the operating system, along with the latest Windows updates available from Microsoft. Some features may require Windows 10 or later versions of the platform. When this is the case, it will be noted in the documentation.

Windows XP and Windows Vista are no longer supported. SocketTools is designed for Windows 7 as the minimum operating system version and will not work correctly on earlier versions of Windows. Although Windows 7 and Windows 8 are no longer supported by Microsoft, SocketTools components will continue to function on those platforms.

Library Redistribution

For applications created using one or more SocketTools libraries, the DLL must be distributed along with the application. The library has no external dependencies, other than standard Windows libraries that are part of the base operating system. In particular, the SocketTools libraries do not use the Microsoft Foundation Classes, nor do they require the Visual C++ Runtime library. The libraries are standard Windows DLLs and do not require COM registration.

If SocketTools is installed on a 64-bit version of Windows, the 32-bit DLLs are installed in the C:\Windows\Syswow64 folder and the 64-bit DLLs are installed in the C:\Windows\System32 folder. When redistributing one or more of the libraries, it is important to make sure that you are selecting the correct version, which is determined by the development tool used and the target platform. For example, if you are using Visual Studio 2019 and target the Win32 platform, then you should only redistribute the 32-bit DLL, regardless if the target system is the 32-bit or 64-bit version of Windows. This is because 32-bit programs can only reference 32-bit libraries. When the application is installed on 64-bit Windows, it will be executed by the WoW64 subsystem which provides a 32-bit environment for the application.

Version Information

The SocketTools libraries have information embedded in them which provides version information to an installation utility. This information called the version resource, specifies the version number as well as other information about the library. If you are using a third-party or in-house installation program, it is extremely important that the program knows how to use this information.

For example, if you are deploying an application which uses the File Transfer Protocol library, the setup program must determine if that library has already been installed on the target system. If it has, it must compare the version resource information in the two libraries. It should only overwrite the library if the version that you have included with your application is later than the one installed on the system. An installation program which overwrites the library without checking the version number may cause the application to fail unexpectedly on the end user's system.

Installation Directory

It is recommended that you install the SocketTools libraries in the same folder with the application that uses them. It is not recommended that you install the libraries under the Windows system folder on an end-user system. If you choose to install the libraries in the Windows system folder, you must ensure that the installer

makes the appropriate registry entries to indicate that they are shared files. Failure to do so can result in the libraries being removed if the user uninstalls your application, which may cause other applications to fail.

If your installer package creates a 32-bit executable and you're deploying a 64-bit application, the installer must be capable of detecting that it is running on a 64-bit system and can disable filesystem redirection to ensure that the 64-bit libraries are installed in the correct location. Consult the documentation for your installer to determine if it is 64-bit compatible.

Windows Install Packages

To help simplify deployment, SocketTools includes MSI (Windows Installer) packages you can use to install the SocketTools libraries on end-user systems. These packages are found in the **Redist** folder where you've installed SocketTools.

Package Name	Description
cstools12_library_x86.msi	SocketTools 12 redistributable libraries for 32-bit applications. This installer is what developers should use if they are targeting the x86 platform and want their software to run on both 32-bit and 64-bit versions of Windows.
cstools12_library_x64.msi	SocketTools 12 redistributable libraries for 64-bit applications. This installer should only be used if 64-bit development tools were used to build the application, and can only be installed on 64-bit versions of Windows.

If you're redistributing a 32-bit application, then all you need is the x86 installer package. If you're redistributing a 64-bit application, then you need the x64 installer package. The installer packages will make sure the libraries are installed in the correct shared Windows folders and will perform the appropriate version checking.

If you have your own installer for your software, then you can redistribute those MSI packages with your installation and use the **msiexec** command to perform the installation. For example, this would install and register the 32-bit libraries with no UI displayed:

```
msiexec /qn /I cstools12_library_x86.msi
```

For the complete list of command line options for **msiexec**, refer to the Windows App Development documentation.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Technical Support

Catalyst Development is committed to providing quality technical support for our products and we offer several different support options designed to meet the needs of our customers. Technical support by email is available for installation, development and redistribution issues related to the purchased product. There are also paid support options available for customers who require additional assistance.

Standard Support

Registered developers have access to a variety of free technical support resources and we always encourage developers to review our online documentation and knowledge base to determine if the question has already been answered.

Frequently Asked Questions

A collection of answers to the most frequently asked questions about a product. General questions about features, functionality and platform compatibility are answered here. The product FAQ is also recommended reading for any developer who is evaluating our software.

Knowledge Base

A searchable online database of solutions to hundreds of common technical questions and problems. The articles provide detailed information, including background information, workarounds and the availability of updates to resolve the problem. This is the first place that most developers should check to determine if the question or problem that they're having has already been addressed.

Online Documentation

A comprehensive collection of online help for our products. Our online help is useful to evaluators who are interested in learning about how our components work and for developers who would like access to the most current reference material.

Release Notes

Information about the latest changes, improvements and corrections made to the current version SocketTools. The release notes can reflect changes that affect all SocketTools editions, as well as updates to a component in a specific edition. If you are upgrading from a previous release, it's recommended that you review the release notes.

Priority Support

For developers who require additional support, Priority Support offers a guaranteed, priority response to technical support issues on the same business day. Corrections which require a source code change and/or documentation change to resolve a problem will be made available as a hotfix at no additional charge, and whenever there is a new product update or hotfix, you will be automatically notified by email.

Premium Support

For developers who have critical support needs, an annual Premium Support agreement offers priority email support and a guaranteed four hour response time during business hours. This support option also includes all of the other benefits of priority support, including hotfixes, source code analysis and assistance with example code. In addition, Premium Support also includes free upgrades if a new version of the product is released while your support agreement is active, ensuring that you're always working with the latest version.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools License Agreement

This License Agreement ("Agreement") is a legal agreement between you and Catalyst Development Corporation ("Catalyst") governing your use of the SocketTools software development kit, including its libraries, controls, assemblies, documentation, sample code, runtime components, and related materials (collectively, the "Software").

"Runtime components" means those portions of the Software intended by Catalyst for redistribution with software applications developed using the Software, including, without limitation, dynamic link libraries (DLLs), ActiveX controls, .NET assemblies, and similar runtime files or components, as identified in the Software, its documentation, or accompanying materials.

In this Agreement, "you" means the individual or legal entity that acquires or uses the Software. A "Developer" is an individual authorized by you to directly use the Software to design, develop, build, maintain, test, or support software applications that incorporate the Software.

By installing, copying, accessing, or using the Software, you agree to be bound by the terms of this Agreement.

1. LICENSE GRANT

Subject to the terms of this Agreement, Catalyst grants you a personal, non-exclusive, non-transferable license to permit a single Developer to install and use the Software for the purpose of designing, developing, testing, maintaining, and supporting software applications.

Each license is assigned to a single Developer, and only that licensed Developer may use the Software.

A separate license is required for each Developer who uses the Software.

A licensed Developer may install and use the Software on any number of computers, virtual machines, cloud-hosted development systems, and other devices under that Developer's control, provided the Software is used solely by that licensed Developer.

The Software may be used to develop both commercial and non-commercial applications.

You may distribute software applications developed using the Software. You may also modify and distribute sample code provided with the Software and distribute the runtime components of the Software, in each case as permitted under this Agreement. However, license keys, registration information, and other licensing credentials are personal to the licensed Developer and may not be shared with others.

Purchase of a license does not entitle you to future versions, upgrades, maintenance plans, subscription services, or support unless separately specified.

2. EVALUATION USE

The evaluation version is provided solely to allow prospective customers to determine whether the Software meets their requirements prior to purchasing a license.

If the Software is provided as an evaluation version, you may use it solely for internal evaluation, testing, and review purposes for up to thirty (30) days.

The evaluation version may not be used for commercial deployment, production use, or redistribution as part of any software application.

Continued use after the evaluation period requires the purchase of a valid license.

The evaluation version may contain technical restrictions, including limitations that prevent redistribution or deployment of applications developed using the evaluation version.

3. REDISTRIBUTION RIGHTS

Subject to the terms of this Agreement, Catalyst grants you a royalty-free right to reproduce and distribute the

runtime components of the Software solely as part of your own software applications.

You may:

- Use the Software in an unlimited number of software projects.
- Redistribute the runtime components with your software applications without additional runtime, deployment, server, user, subscription, royalty, or project-based licensing fees.
- Modify and redistribute sample code provided with the Software as part of your software applications.

You may not:

- Redistribute the Software itself as a software development kit, library collection, development tool, hosting platform, or competing product.
- Permit third parties to use the Software as a substitute for obtaining their own Developer license.
- Misrepresent the ownership or origin of the Software.

Pre-release versions of the Software provided for evaluation purposes may be subject to additional terms and may not be suitable for production use. Such versions may contain defects, incomplete functionality, or other issues which are not present in released versions. Any deployment, redistribution, or production use of pre-release software is undertaken solely at your own risk.

4. LICENSE TRANSFERS

If you are an organization, you may permanently reassign a license from one Developer to another, provided the original Developer permanently ceases all use of the Software.

Licenses may not be shared among multiple Developers or reassigned on a temporary or rotating basis.

At any given time, a separate license is required for each Developer actively using the Software.

5. OWNERSHIP

The Software is licensed, not sold.

Catalyst and its licensors retain all right, title, and interest in and to the Software, including all copyrights, trademarks, trade secrets, and other intellectual property rights.

All rights not expressly granted under this Agreement are reserved by Catalyst and its licensors.

Licenses granted under this Agreement are perpetual unless you materially violate this Agreement and do not correct the violation within thirty (30) days after written notice from Catalyst. If a license ends for that reason, you must stop using the Software and remove or destroy copies of it in your possession or control, except that software applications previously developed and lawfully distributed in compliance with this Agreement may continue to be distributed and supported.

6. THIRD-PARTY SERVICES

Certain features of the Software may enable access to third-party services, including artificial intelligence providers, cloud platforms, databases, application programming interfaces (APIs), search services, and other services operated by entities other than Catalyst ("Third-Party Services").

Use of any Third-Party Service is governed solely by the terms, policies, pricing, and requirements established by the applicable service provider.

You acknowledge and agree that:

- You are solely responsible for obtaining and maintaining any accounts, subscriptions, credentials, API

keys, licenses, or permissions required to access Third-Party Services.

- You are solely responsible for all fees, charges, usage costs, token consumption charges, subscription fees, overage fees, and other expenses associated with Third-Party Services.
- Catalyst does not act as a reseller, billing intermediary, payment processor, or agent for any Third-Party Service provider.
- Any agreement regarding pricing, billing, service levels, quotas, credits, limitations, or availability exists solely between you and the applicable Third-Party Service provider.
- Catalyst does not control and is not responsible for the availability, operation, pricing, content, security, policies, or future compatibility of any Third-Party Service.
- Third-Party Services may be modified, restricted, suspended, rate-limited, or discontinued at any time without notice.
- Catalyst is not responsible for the content, responses, recommendations, generated output, decisions, or other information produced by any Third-Party Service.

7. CATALYST SERVICES

Certain features of the Software may rely on services operated by Catalyst.

Catalyst may provide online services, hosted APIs, cloud storage services, update services, location services, or other resources related to the Software.

Catalyst may modify, suspend, limit, or discontinue such services at any time. Access to such services may be subject to separate usage terms, service plans, published usage limits, or technical restrictions.

Nothing in this Agreement guarantees the continued availability of any online service or any future software update.

8. WARRANTY DISCLAIMER

THE SOFTWARE IS PROVIDED "AS IS".

TO THE MAXIMUM EXTENT PERMITTED BY LAW, CATALYST DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ACCURACY, AND TITLE.

CATALYST DOES NOT WARRANT THAT THE SOFTWARE WILL OPERATE WITHOUT INTERRUPTION OR ERROR.

9. LIMITATION OF LIABILITY

TO THE MAXIMUM EXTENT PERMITTED BY LAW, CATALYST SHALL NOT BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, EXEMPLARY, OR PUNITIVE DAMAGES, INCLUDING LOST PROFITS, LOST DATA, BUSINESS INTERRUPTION, LOSS OF GOODWILL, OR COSTS OF SUBSTITUTE GOODS OR SERVICES.

IN NO EVENT SHALL CATALYST'S TOTAL LIABILITY ARISING OUT OF OR RELATED TO THIS AGREEMENT EXCEED THE AMOUNT PAID FOR THE SOFTWARE LICENSE GIVING RISE TO THE CLAIM.

10. EXPORT COMPLIANCE

You agree to comply with all applicable export control laws and regulations. You may not export, re-export, transfer, or provide access to the Software in violation of such laws.

11. GOVERNING LAW

This Agreement is governed by the laws of the State of California, excluding its conflict of laws principles.

Any legal action arising out of or relating to this Agreement must be brought in a court of competent jurisdiction located in California, and the parties consent to the jurisdiction of those courts.

12. GENERAL

This Agreement constitutes the entire agreement between the parties concerning the Software and supersedes all prior or contemporaneous agreements and understandings relating to the Software.

If any provision of this Agreement is found to be unenforceable, the remaining provisions will remain in full force and effect.

If Catalyst does not enforce a provision of this Agreement, that does not waive its right to enforce it later.

SocketTools, SocketWrench and related names are trademarks of Catalyst Development Corporation. All other trademarks are the property of their respective owners.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Copyright Information

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Catalyst Development Corporation™, SocketTools™ and SocketWrench™ are trademarks of Catalyst Development Corporation. Microsoft™, Windows™, Visual Basic™ and Visual Studio™ are trademarks or registered trademarks of Microsoft Corporation.

Portions Copyright © 1993, 1994 The Regents of the University of California
Portions Copyright © 1989 Massachusetts Institute of Technology
Portions Copyright © 1995 Tatu Ylonen
Portions Copyright © 1999, 2000 Neil Provos and Markus Friedl
Portions Copyright © 1995, 2005 Jean-loup Gailly and Mark Adler
Portions Copyright © 2004-2007 Sara Golemon
Portions Copyright © 2005,2006 Mikhail Gusarov
Portions Copyright © 2006-2007 The Written Word, Inc.
Portions Copyright © 2007 Eli Fant
Portions Copyright © 2009-2023 Daniel Stenberg
Portions Copyright © 2008, 2009 Simon Josefsson
Portions Copyright © 2000 Markus Friedl
Portions Copyright © 1998-2023 OpenSSL
Portions Copyright © 1991, 1992 RSA Data Security, Inc.
Portions Copyright © 2015 Microsoft Corporation

Information in this document is subject to change without notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Catalyst Development Corporation. Portions of SocketTools uses open source software subject to the MIT and Apache licenses.

The software described in this document is furnished under a license agreement. The software may be used only in accordance with the terms of the agreement. It is against the law to copy the software except as specifically allowed in the license agreement. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without the express written permission of Catalyst Development Corporation.

SocketTools 12 Library Edition Developers Guide

- Introduction
    1. Features
    2. Getting Started
- General Concepts
    1. Protocols
    2. Transmission Control Protocol
    3. User Datagram Protocol
    4. Domain Names
    5. Service Ports
    6. Sockets
    7. Handles
    8. Security Protocols
    9. Digital Certificates
- Development Overview
    - Application Design
    - Program Structure
    - Library Initialization
    - Asynchronous Connections
    - Secure Connections
    - Network Input/Output
    - Event Handling
    - Error Handling
    - Debugging Facilities
- Language Support
    1. Data Types
    2. Unicode
    3. Visual C++
    4. Visual C#
    5. Visual Basic .NET
    6. Visual Basic 6.0
    7. Delphi
    8. PowerBASIC
- Library Overview
    - Domain Name Service
    - File Encoding and Compression
    - File Transfer Protocol
    - GeoIP Location
    - Hypertext Transfer Protocol
    - Internet Control Message Protocol
    -

Internet Message Access Protocol

- Large Language Model Client
- Mail Message (MIME) Parser
- Network News Transfer Protocol
- Network Time Protocol
- News Feed
- Post Office Protocol
- Remote Command Protocol
- Simple Mail Transfer Protocol
- Telnet Protocol
- Terminal Emulation
- Text Messaging
- Web Storage
- WebView Browser
- Whois Protocol
- Windows Sockets (SocketWrench)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Features

The SocketTools libraries can be used in a wide variety of programming languages, including Visual C++, Visual C#, Visual Basic.NET and Visual Basic 6.0, as well as C++ Builder, Delphi and a variety of other development environments. Any language which is capable of calling functions exported from a Windows dynamic link library can take advantage of the SocketTools Library Edition.

Features of the SocketTools Library Edition include:

- An efficient architecture designed to reduce the overhead of using other types of components such as ActiveX controls. The libraries in the SocketTools Library Edition are not COM libraries, but rather standard Windows dynamic link libraries which have been optimized for high performance and low resource utilization on the Windows platform.
- There are no external dependencies on third party libraries or components, and each DLL is completely self-contained. We do not require that you redistribute large shared libraries like the Microsoft Foundation Classes or Visual C++ runtime libraries. Not only does this make redistribution of your software easier, it can reduce the overall footprint for applications which do not need to use these libraries themselves.
- An interface which is designed for broad-based compatibility with a variety of programming languages, not just for C or C++ programmers. You won't see functions that only provide complex interfaces, using data types or structures which are difficult or impossible to represent in other programming languages. Following the model of the Windows API, the functions use handles (integer values) to reference client sessions and most data types used as function parameters are null-terminated strings, integers or byte arrays. In those cases where structures are used, they are designed to be compatible with most languages. Simply put, if your programming language can call functions in the Windows API, you can use the SocketTools Library Edition.
- A comprehensive design which supports both high-level operations as well as lower-level functions at the protocol level. For example, the File Transfer Protocol library has functions such as **FtpPutFile** and **FtpGetFile** which allow an application to easily upload and download files in a single function call. It also includes lower-level functions like **FtpOpenFile** to open a file on the server and access it in a fashion similar to traditional file I/O operations.
- Support for both synchronous (blocking) and asynchronous (non-blocking) operation depending on the needs of the application. Asynchronous operation is supported by an event-driven model where the application is notified of networking events by user-defined messages posted to the message queue. Event notification can be enabled, disabled, suspended and resumed completely under the control of the application, giving developers complete freedom in controlling their behavior of their software. Synchronous operation is also fully supported, enabling developers to easily write programs in a procedural "top down" programming style without using an event-driven approach.
- Support for function callbacks during high-level synchronous operations, such as downloading a file or sending an email message. This allows an application to make changes to its user interface, such as updating a progress bar. This enables the developer to take advantage of the simplicity of using high-level functions without sacrificing the flexibility or features expected by the user.
- The Library Edition enables applications to take advantage of security features, such as support for the transport Layer Security (TLS) protocol and 256-bit encryption. The libraries use the Windows CryptoAPI to provide security services, which means there are no third-party security libraries that must be installed by your users. Taking advantage of the security features in the SocketTools Library Edition is as simple as setting a few API options before connecting to the server. The protocol negotiation, data encryption and decryption is handled transparently by the library. From the perspective of the application developer, it is just as if it were a standard connection to the server.

- Libraries which are thread-safe and optimized for applications which use multiple threads. The SocketTools libraries fully support multithreaded applications and implement an internal architecture that insures that client sessions can be safely created and used by multiple threads. Applications can create worker threads and pass client handles to those threads to perform some function and then return the handle back to the original owner or simply terminate the connection.

The SocketTools Library Edition includes everything professional software developers need to create complex programs that take advantage of the standard Internet protocols, enabling developers to focus on their core application technology rather than the details of how to upload a file or retrieve an email message from a server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Getting Started

SocketTools is a large collection of libraries that can be used to create a variety of applications, so deciding what protocols and libraries you'll need to use will be the first step. SocketTools covers several general categories, and there is some cross-over between libraries in terms of functionality. We'll cover the most common programming needs and discuss what protocols should be used. Note that this section doesn't cover all of the libraries in SocketTools, and more specific information for each library is available within the toolkit.

One thing you'll discover as you start to use SocketTools is that the API was intentionally designed to be consistent between many of the libraries. For example, both the File Transfer Protocol and Hypertext Transfer Protocol libraries can be used to upload and download files, and the functions for both of those libraries are very similar. Once you've become comfortable working with one of the libraries, you'll find it very easy to use the other, related libraries.

File Transfers

- File Transfer Protocol
- Hypertext Transfer Protocol

One of the most common requirements for an application is the ability to upload and download files, either over the Internet or between systems on a local intranet. There are two core protocols which are used for file transfers, the File Transfer Protocol (FTP) and the Hypertext Transfer Protocol (HTTP). The decision as to which protocol to use largely depends on whether or not the program must also perform any type of file management on the server. Because many of the functions in the FTP and HTTP APIs are similar, you may wish to use both and simply give your users an option as to which protocol they prefer to use.

If your program needs to upload files or manage the files on the server, we recommend that you use FTP. In addition to uploading and downloading files, FTP can be used to rename or delete files, create directories, list the files in a directory and perform a variety of other functions. On the other hand, if you primarily need to just download files, HTTP can be a better choice. The protocol is simpler and you're less likely to encounter some of the issues that can arise when using FTP from behind a firewall.

It is also an option to use FTP to upload and manage files and HTTP to download files within the same program. The important thing to keep in mind is that if you want to use HTTP and need to upload files, you must make sure that the server has been configured for it. Most web servers do not support the ability to upload files by default; it requires the administrator to specifically enable that functionality.

World Wide Web

- Hypertext Transfer Protocol

If you need to access documents or execute scripts on a web server, you'll want to use the Hypertext Transfer Protocol (HTTP) library. You can use the library to download files and post data to scripts. The library also supports the ability to upload files, either using the PUT command or by using the POST command, which is the same method used when selecting a file to upload using a form. The library can also be used to execute custom commands, allowing your application to take advantage of features like WebDAV, a distributed authoring extension to HTTP.

Web Services

- Application Storage Service
- GeoIP Location Service
- Text Messaging Service

SocketTools provides secure services that you can use to store and manage application data, obtain geographical location information based on an IP address, and send text messages through SMS gateways. These services use secure, encrypted connections to our servers, and our Web Services library provides a high-level interface to the underlying RESTful service APIs which are used.

Electronic Mail

- Domain Name Services Protocol
- Internet Message Access Protocol
- Mail Message Library
- Post Office Protocol
- Simple Mail Transfer Protocol

There are a number of SocketTools libraries which can be used by an application that needs to send email messages or retrieve them from a user's mailbox. The email related libraries can be broken into three groups, those that deal primarily with managing and retrieving messages for a user, those which are used to send messages and those which can be used for either purpose.

The two principal protocols used to manage a user's email are the Post Office Protocol (POP3) and the Internet Message Access Protocol (IMAP). POP3 is the protocol that the majority of Internet Service Providers (ISP) use to give their customers access to their messages. It is primarily designed to enable an application to download the messages from the mail server and store them on the local system. Once all of the messages have been downloaded, they are deleted from the server. The user's mailbox is essentially treated as a temporary storage area.

On the other hand, IMAP is designed to allow the application to manage the messages on the server. You can create new mailboxes, move messages between mailboxes and search for messages. Because IMAP can be used to access specific parts of a message, it's not necessary to download the entire message if you just want to read a specific part of it. In terms of the SocketTools APIs, it's useful to think of the functions in the IMAP library as a superset of those in the POP3 library. You'll find that functions used for accessing messages are very similar, but the IMAP library contains additional functions for managing mailboxes and performing operations that are specific to that protocol, such as the ability to search for messages.

To send an email message to someone, the protocol that you'll use is the Simple Mail Transfer Protocol (SMTP). The SocketTools library supports the standard implementation of this protocol, along with many of the extensions that have been added since its original design. Extended SMTP (ESMTP) provides features such as authentication, delivery status notification, secure connections using TLS and so on. Another library that you may use is the Domain Name Services (DNS) library, which your application can use to determine what servers are responsible for accepting mail for a particular user.

Common to both sending and receiving email messages is the need to be able to create and process those messages. An email message has a specific structure which is defined by a number of standards, collectively called the Multipurpose Internet Mail Extensions (MIME). The SocketTools Mail Message library can be used to create messages in the format, as well as parse existing messages so that you application can access the specific information that it needs. For example, you can use this library to attach files to a message as well as extract a specific file attachment from a message and store it on the local system.

Terminal Sessions

- Rlogin Protocol
- Secure Shell Protocol
- Telnet Protocol
- Terminal Emulation

If you need to establish an interactive terminal session with a server, there are two protocols that you can use. The most common is the Telnet Protocol; however, there is also the Rlogin protocol which is part of the Remote Command library. Either of these protocols are typically used in conjunction with the Terminal Emulation library, which provides ANSI and DEC VT-220 terminal emulation functions. Used together, the user can login and interact with the server in the same way that they would use a console or character based terminal.

Newsgroups

- File Encoding Library
- Mail Message Library
- Network News Transfer Protocol

If you need to access newsgroups, the Network News Transfer Protocol will enable you to connect, list, retrieve and post articles. Because news articles have a format that is very similar to email messages, the Mail Message library can be used to parse articles that you've downloaded or create new articles to be posted. If you need to attach a file to the article that you're posting, the File Encoding library can be used to encode the file using the yEnc encoding algorithm, which has become the de facto standard on USENET.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Examples

The examples provided in this documentation are written in ANSI C or standard C++, using conventions that align closely with the Windows SDK. While the majority of SocketTools developers use C or C++, support is also provided for other languages such as Visual Basic, Pascal, and .NET (C# and VB.NET). Where applicable, language-specific notes and P/Invoke declarations are available to simplify integration.

Language Requirements

The code samples are written in standard C and C++ and assume a 32-bit or 64-bit Windows environment. The minimum C language standard used is **C99**. While no specific features from C11 or later are required, all development and testing is performed with modern toolchains. The SocketTools header file is designed to be compatible with most modern C/C++ development tools.

C++ examples use language features compatible with the Microsoft compiler and are structured for compatibility with the class-based wrappers provided in the SocketTools libraries. Some examples use Microsoft Foundation Classes (MFC) or the Active Template Library (ATL) which provides high-level classes designed to simplify C++ development on the Windows platform.

Naming Conventions

The SocketTools API follows naming conventions which are similar to the Windows API:

- Pointers are typically prefixed with `p`, `lp`, or `psz` depending on type.
- Parameters for null-terminated strings typically use `lpsz` or `psz` as a prefix.
- Boolean values use `BOOL` and constants such as `TRUE` and `FALSE`.
- String arguments use standard Windows types like `LPCSTR`, `LPWSTR`, or `LPTSTR`.
- Functions are named using PascalCase such as `InetConnect` and `HttpConnect`.

This consistent naming scheme is intended to reduce friction for Windows developers who are already familiar with the Win32 API and associated SDKs. Developers using other languages or platforms may choose to follow alternate naming conventions when writing their own wrapper functions.

Unicode Support

Many of the examples use `TCHAR` and the `_T()` macro to support both ANSI and Unicode builds from the same codebase. This is part of the standard Windows SDK and is defined in the header file `<tchar.h>`. When the `UNICODE` macro is defined, `TCHAR` resolves to `wchar_t` and string literals are UTF-16 strings. When it is not defined, `TCHAR` resolves to an 8-bit `char` type. If you are unfamiliar with these macros, or are working in a language that does not support them, it is generally safely to replace `TCHAR` with whatever represents a character in your language.

It is strongly recommended that developers build their applications with Unicode support enabled. Unicode builds are the default configuration for modern Microsoft compilers, including Visual Studio, and offer the broadest compatibility across languages and locales. By using Unicode, applications can handle multilingual content without relying on code page settings or locale-specific behavior. This is especially important for Internet applications where content may originate from systems using different languages and character sets.

For .NET developers, all P/Invoke declarations provided for C# and VB.NET are based on the Unicode versions of the API functions. This approach aligns with the .NET runtime's use of UTF-16 strings and eliminates the need for manual encoding or conversion. It ensures consistent behavior and simplifies interop with native code. Developers targeting .NET should avoid the ANSI versions of the API and use the Unicode declarations provided for the best results.

ANSI and UTF-8

SocketTools provides both ANSI and Unicode versions of most functions, similar to the Windows API. Functions with names ending in **A** accept narrow (8-bit) strings, while those ending in **W** accept wide (UTF-16). Functions without a suffix are either binary-safe or use only numeric types. For example, functions which expect pointers to byte (unsigned character) arrays will not have separate versions.

Unlike the Windows API, most ANSI functions in SocketTools expect UTF-8 encoded text rather than legacy multibyte character sets (MBCS) such as Windows-1252 or Shift-JIS. This ensures consistent behavior across systems with different regional settings and allows interoperability with modern services that expect UTF-8 as the default encoding.

Microsoft supports UTF-8 as a preferred character encoding in the Windows API. With Windows 10 and later versions, applications can opt into using UTF-8 as their default code page by calling **SetThreadLocale** with code page **CP_UTF8** or by setting a manifest option. SocketTools embraces this approach by treating UTF-8 as the standard encoding for all narrow string APIs. This design allows the SocketTools functions to avoid regional code page ambiguities, improves compatibility with international services, and simplifies handling of mixed-language content. Developers can use UTF-8 directly in their applications without worrying about locale-specific behavior.

Some programming environments do not offer full support for Unicode APIs, particularly legacy languages such as Visual Basic 6. In those cases, developers should use the ANSI versions of the SocketTools API functions, which accept UTF-8 encoded strings. Although UTF-8 is technically Unicode, it is compatible with standard 8-bit string handling in those languages and allows applications to work with international content more reliably than using legacy code pages. When using the ANSI API, developers are responsible for ensuring that strings passed to the API are properly encoded as UTF-8. In most cases, this involves converting from the native character encoding used by the language into UTF-8 prior to calling the function. This approach provides a practical bridge for working with modern services and data formats in environments that lack native Unicode support.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

General Concepts

This section of the developer's guide will cover the core networking protocols along with the general concepts related to Internet programming. Although it is not necessary to understand the lower level details of network programming in order to use SocketTools, it is useful to be familiar with the basic concepts and terminology.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Protocol Standards

Throughout this documentation, the term protocol refers to a set of rules that define how computers communicate with one another. There are two types of protocols which are important when working with network applications: transport protocols and application protocols.

Transport protocols define how data is exchanged between systems. The two transport protocols discussed in this guide are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). These protocols provide the underlying communication channel used by network applications.

Application protocols are built on top of transport protocols to perform specific tasks. For example, the File Transfer Protocol (FTP) is used to transfer files, while the Simple Mail Transfer Protocol (SMTP) is used to send email messages. Other common application protocols include HTTP for web services and SSH for secure remote access.

The Internet application protocols are defined in standards documents called RFCs (Request For Comments) which are maintained by the Internet Engineering Task Force. The following protocols standards are implemented by the SocketTools libraries:

File Transfer

Protocol	Standards	Description
FTP	RFC 959, RFC 1579, RFC 2228, RFC 2428, RFC 3659	File Transfer Protocol including security, NAT/firewall support, and extended directory services.
SSH / SFTP	RFC 4251, RFC 4252, RFC 4253, RFC 4254	Secure Shell protocol used for secure remote access and file transfers.
TLS	RFC 8446	Transport Layer Security used by FTPS and other secure protocols.

Web Services and REST APIs

Protocol	Standards	Description
HTTP	RFC 9110, RFC 9112 ***Legacy:*** RFC 1945, RFC 2616	Hypertext Transfer Protocol used by websites, REST APIs and cloud services.
URI / URL	RFC 3986 ***Legacy:*** RFC 1738	Uniform Resource Identifier syntax and resource naming conventions.
TLS	RFC 8446	Secure transport protocol used by HTTPS services and APIs.
Compression	RFC 1951, RFC 1952	Deflate and GZIP compression formats commonly used by HTTP servers.

Electronic Mail

Protocol	Standards	Description
SMTP	RFC 5321, RFC 1869, RFC 6531 ***Legacy:*** RFC 2821	Message submission and transfer protocol with service extensions and internationalized email support.
Message Format	RFC 5322	Internet message header and body format.

	Legacy: RFC 822	
MIME	RFC 2045, RFC 2049, RFC 2183, RFC 2231	Multipurpose Internet Mail Extensions including attachments, content types and parameter encoding.
POP3	RFC 1939	Post Office Protocol Version 3.
IMAP4	RFC 3501	Internet Message Access Protocol Version 4.
TLS	RFC 8446	Secure transport protocol used by SMTPS, POP3S and IMAPS.

Network News (Usenet)

Protocol	Standards	Description
NNTP	RFC 3977 *Legacy:* RFC 977, RFC 2980	Network News Transfer Protocol used to access and manage Usenet servers.
TLS	RFC 8446	Secure transport protocol used by NNTPS servers.

Internet Infrastructure

Protocol	Standards	Description
DNS	RFC 1034, RFC 1035	Domain Name System used to resolve host names and service records.
ICMP	RFC 792	Internet Control Message Protocol used by Ping and network diagnostics.
IPv6	RFC 8200	Internet Protocol Version 6.
Time Protocol	RFC 868	Simple network time service.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Transmission Control Protocol

When two computers wish to exchange information over a network, there are several components that must be in place before the data can actually be sent and received. Of course, the physical hardware must exist, which is typically either a network interface card (NIC) or a serial communications port for dial-up networking connections. Beyond this physical connection, however, computers also need to use a protocol which defines the parameters of the communication between them. In short, a protocol defines the "rules of the road" that each computer must follow so that all of the systems in the network can exchange data. One of the most popular protocols in use today is TCP/IP, which stands for Transmission Control Protocol/Internet Protocol.

By convention, TCP/IP is used to refer to a suite of protocols, all based on the Internet Protocol (IP). Unlike a single local network, where every system is directly connected to each other, an internet is a collection of networks, combined into a single, virtual network. The Internet Protocol provides the means by which any system on any network can communicate with another as easily as if they were on the same physical network. Each system, commonly referred to as a host, is assigned a numeric value which can be used to identify it over the network. These numeric values are known as IP addresses, and are usually represented as a string value that contains a series of numbers.

There are two versions of TCP/IP and two different IP address formats based on which version of the protocol is being used. For Internet Protocol v4 (IPv4), addresses are 32 bits wide and are represented by a sequence of four 8-bit numbers separated by periods. This is called dot-notation and looks something like **192.168.19.64**. This is the address format that many developers are familiar with because IPv4 continues to be the most commonly used version of the protocol. Internet Protocol v6 (IPv6) is the next generation of IP and it supports a much larger address space as well as a number of other features. IPv6 addresses are 128 bits wide and represented by a sequence of hexadecimal values separated by colons. As expected, this format is much longer than the simple dot-notation used by IPv4 address. A typical IPv6 address will look something like **fd7c:2f6a:4f4f:ba34::a32**, although there are certain shorthand notations that can be used. SocketTools supports both IPv4 and IPv6, and can automatically determine which version of the protocol should be used based on the address. Because IPv4 is still widely used, if given a choice between using IPv4 or IPv6, the SocketTools components will choose IPv4 for backwards compatibility whenever possible. However, an application can choose to exclusively use IPv6 if required.

When a system sends data over the network using the Internet Protocol, it is sent in discrete units called datagrams, also commonly referred to as packets. A datagram consists of a header followed by application-defined data. The header contains the addressing information which is used to deliver the datagram to its destination, much like an envelope is used to address and contain postal mail. And like postal mail, there is no guarantee that a datagram will actually arrive at its destination. In fact, datagrams may be lost, duplicated or delivered out of order during their travels over the network. Needless to say, this kind of unreliability can cause a lot of problems for software developers. What's really needed is a reliable, straightforward way to exchange data without having to worry about lost packets or mixed data.

To fill this need, the Transmission Control Protocol (TCP) was developed. Built on top of IP, TCP offers a reliable, full-duplex byte stream which may be read and written to in a fashion similar to reading and writing a file. The advantages to this are obvious: the application programmer doesn't need to write code to handle dropped or out-of-order datagrams, and instead can focus on the application itself. And because the data is presented as a stream of bytes, existing code can be easily adopted and modified to use TCP.

TCP is known as a connection-oriented protocol. In other words, before two programs can begin to exchange data they must establish a connection with each other. This is done with a three-way handshake in which both sides exchange packets and establishes the initial packet sequence numbers. The sequence number is important because, as mentioned above, datagrams can arrive out of order; this number is used to ensure that data is received in the order that it was sent. When establishing a connection, one program must assume the role of the client, and the other the server. The client is responsible for initiating the connection, while the

server's responsibility is to wait, listen and respond to incoming connections. Once the connection has been established, both sides may send and receive data until the connection is terminated.

Most of the application protocols which are supported by SocketTools use TCP to communicate over the Internet or local intranet. However, it is important to remember that it is not necessary for you to understand how TCP/IP works at the lowest levels in order to use SocketTools. Complex operations such as performing checksums on packets of data to ensure they arrive intact are handled for you automatically. In most cases, the SocketTools API provides functions which are similar to what you would use when reading or writing to a file.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

User Datagram Protocol

Unlike TCP, the User Datagram Protocol (UDP) does not present data as a stream of bytes, nor does it require that you establish a connection with another program in order to exchange information. Data is exchanged in discrete units called datagrams, which are similar to IP datagrams. In fact, the only features that UDP offers over raw IP datagrams are port numbers and an optional checksum.

UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at its destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP. Much of the work that TCP does transparently (such as generating checksums, acknowledging the receipt of packets, retransmitting lost packets and so on) must be performed by the application itself.

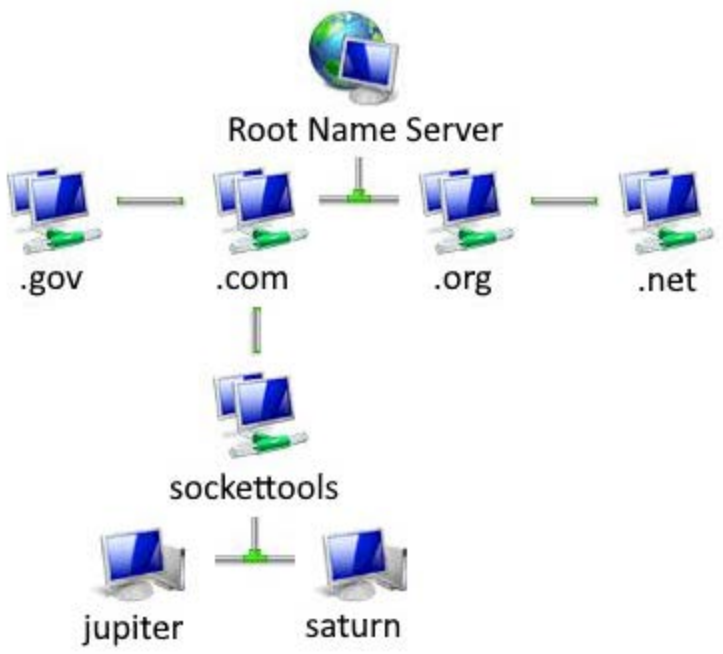
With the limitations of UDP, you might wonder why it's used at all. UDP has the advantage over TCP in two critical areas: speed and packet overhead. Because TCP is a reliable protocol, it goes through great lengths to ensure that data arrives at its destination intact, and as a result it exchanges a fairly high number of packets over the network. UDP doesn't have this overhead, and is considerably faster than TCP. In those situations where speed is paramount, or the number of packets sent over the network must be kept to a minimum, UDP is the solution.

A few of the SocketTools libraries use UDP as the method of communicating with a server. The Domain Name Services library and the Network Time Protocol library both use UDP to request information from a server. The amount of data exchanged is typically very small, and UDP is well suited for those protocols. In addition, the Internet Control Message Protocol uses a special type of IP datagram in order to determine information about a server, such as whether it is reachable and the amount of time that it takes to exchange data with the local system. More information about these protocols will be presented later in the Developer's Guide.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Names

An application must have several pieces of information to exchange data with a program running on another system. The first is the Internet Protocol (IP) address of the computer system on which the other program is running. Although this address is internally represented by a numeric value (either 32 or 127 bits wide), it is typically identified by a logical name called a host name or fully qualified domain name. Host names are divided into several parts separated by periods, called domains. The structure is hierarchical, with the top-level domains defining the type of organization that network belongs to, and sub-domains further identifying the specific network. Everyone who has used a web browser is familiar with host names such as **www.microsoft.com**.



In this figure, the top-level domains are "gov" (government agencies), "com" (commercial organizations), "edu" (educational institutions) and "net" (Internet service providers). The fully qualified domain name is specified by naming the host and each parent sub-domain above it, separating them with periods. For example, the fully qualified domain name for the "jupiter" host would be "jupiter.sockettools.com". In other words, the system "jupiter" is part of the "catalyst" domain (a company's local network) which in turn is part of the "com" domain (a domain used by all commercial enterprises).

To use a host name instead of an IP address to identify a specific system or network, there must be some correlation between the two. This is accomplished by one of two means: a local host table or a name server. A host table is a text file that lists the IP address of a host, followed by the names by which it is known. A name server is a system which can be presented with a host name and will return that host's IP address. This approach is advantageous because the host information for the entire network is maintained in one centralized location, rather than being scattered over every system on the network.

The standard protocol used to convert a host name into an IP address is called the Domain Name Service (DNS) protocol. All of the SocketTools networking libraries have the ability to automatically convert between host names and IP addresses, and in most cases they can be used interchangeably. For example, those functions which require that you specify the name of a server to connect to, you can use either its host name or its IP address. In addition, SocketTools has a library that specifically supports the Domain Name Service protocol, enabling your application to send specialized queries to the name server. Later in the Developer's Guide there will be information about how DNS can be used in a number of different types of applications.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Service Ports

In addition to the IP address of the server, an application also needs to know how to address the specific program that it wishes to communicate with. This is accomplished by specifying a service port, a number between 1 and 65535 that uniquely identifies an application running on the system. A port can be referred to by its number, or by a name that is associated with that number. Like hostnames, service names are usually matched to port numbers through a local file, commonly called services. This file lists the logical service name, followed by the port number and protocol used by the server.

A number of standard service names are used by Internet-based applications and these are referred to as Well Known Services. These services are defined by a standards document and include common application protocols used for transferring files, accessing documents on a webserver or sending and receiving email messages. In most cases, when connecting to a service using the SocketTools libraries, they will default to the appropriate port number for that server. For example, the File Transfer Protocol library has default port values for standard and secure connections. Specifying a different port number is only necessary if you know that the server has been configured to use a non-standard port number.

It is important to remember that a service name or port number is a way to address an application running on a server. Because a particular service name is used, it doesn't guarantee that the service is available, just as dialing a telephone number doesn't guarantee that there is someone at home to answer the call.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Sockets

The previous sections described what information a program needs to communicate over a TCP/IP network. The next step is for the program to create what is called a socket, a communications end-point that can be likened to a telephone. However, creating a socket by itself doesn't let you exchange information, just like having a telephone in your house doesn't mean that you can talk to someone by simply taking it off the hook. You need to establish a connection with the other program, just as you need to dial a telephone number, and to do this you need the address of the application that you want to connect to. This address consists of three key parts: the protocol family, Internet Protocol (IP) address and the service port number.

We've already talked about the IP address and service port, but what's the protocol family? It's a number which is used to logically designate a group of related protocols. Since the socket interface is general enough to be used with several different protocols, the protocol family tells the underlying network software which protocol is being used by the socket. In our case, the Internet Protocol family will always be used when creating sockets. With the protocol family, IP address of the system and the service port number for the program that you want to exchange data with, you're ready to establish a connection.

For the most part, it is not necessary for applications which use the SocketTools libraries to directly make use of the low-level socket interface in order to communicate over the Internet. Instead, SocketTools provides a higher level of abstraction where a connection is identified using a handle to a client session.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Handles

Throughout the documentation for the SocketTools functions, you will see references to handles. Windows developers are generally familiar with the concept of handles since they are used throughout the Windows API. In simplest terms, a handle is an unsigned integer value that is used to represent some object in memory. The actual value of the handle is not important and may refer to the memory address of a specific object, or it may be an index into a table of objects. When a handle is created, its value is unique for the life of each object created by the process. It is important that an application never make assumptions about the specific value of a handle. Handle values may be reused for objects that have been destroyed, and there is no guarantee that handle values are assigned in any particular order.

In SocketTools, handles are commonly used to refer to client sessions. A session begins when the library is used to establish a connection with the server and ends when that connection is terminated. The client handle is defined as an unsigned integer type called HCLIENT and is returned by those functions which create a new connection. When the connection is terminated, the handle is released, along with any system resources that were allocated for it. An unused handle is identified by the constant INVALID_CLIENT. If a function returns this value instead of a valid handle, it indicates that the function has failed. It is important to note that the handles returned by the SocketTools functions are not necessarily socket handles and cannot be used interchangeably with the Windows Sockets API or other Windows kernel functions.

When your application targets the x86 platform, SocketTools handles are 32 bits wide. When you target the x64 platform, the handles are 64 bits wide, however only the lower 32 bits are significant. This means that it is safe to cast a 64-bit SocketTools handle to a 32-bit integer value and then back to a handle if necessary. It is recommended that you always use the appropriate handle type in your code, such as HCLIENT. Assumptions about the width of a handle in your program can lead to portability problems if you ever decide to create a 64-bit version of your application.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Security Protocols

Security and privacy is a concern for everyone who uses the Internet, and the ability to provide secure transactions over the Internet has become one of the key requirements for many business applications. The SocketTools Library Edition has the ability to establish secure connections with servers. Although most of the technical issues such as data encryption are handled internally by the library, a general understanding of the standard security protocols is useful when designing your own applications.

When you establish a connection to a server over the Internet (for example, a web server), the data that you exchange is typically routed over dozens of computer systems until it reaches its destination. Any one of these systems may monitor and log the data that it forwards, and there is no way for either the sender or receiver of that data to know if this has been done. Exchanging information over the Internet could be likened to talking with someone in a public restaurant. Anyone can choose to listen to what you're saying, and unless they introduce themselves, you have no idea who they are or if they've even heard what you said.

To ensure that private information can be securely exchanged over the Internet, two basic requirements must be met: there must be a way to send that information so that only the sender and the receiver can understand what is being exchanged, and there must be a way for them to determine that they each are in fact who they claim to be. The solution to the first problem is to use encryption, where a key is used to encrypt and decrypt the data using a mathematical formula. The second problem is addressed by using digital certificates. These certificates are issued by a certificate authority (CA), which is a trusted third-party organization who verifies the individual or company which is issued a certificate are who they claim to be. These two concepts, encryption and digital certificates, are combined to provide the means to send and receive secure information over the Internet.

The Secure Sockets Layer (SSL) protocol was originally developed by Netscape as a way to exchange information securely over the Internet, and is no longer widely used. Improvements to SSL have resulted in the Transport Layer Security (TLS) protocol, and it has become the the standard for secure communications over the Internet. Both of these protocols are designed to allow a private exchange of encrypted data between the sender and receiver, making it unreadable by an intermediate system. Using the restaurant analogy, it would be as if two people were speaking in a language that only they could understand. Although someone sitting at the next table could listen in on the conversation, they wouldn't have any idea what was actually being said.

A secure connection, for example between a web browser and a server, begins with what is called the handshake phase where the client and server identify themselves. When the client first connects with the server it sends a block of data to the server and the server responds with its digital certificate, along with its public key and information about what type of encryption it would like to use. Next, the client generates a master key and sends this key to the server, which authenticates it. Once the client and server have completed this exchange, keys are generated which are used to encrypt and decrypt the data that is exchanged. With the handshake completed, a secure connection between the client and server is established. SocketTools handles the handshake phase of the secure connection automatically and does not require any additional programming. If a secure connection cannot be established, an error is returned and the network connection is closed.

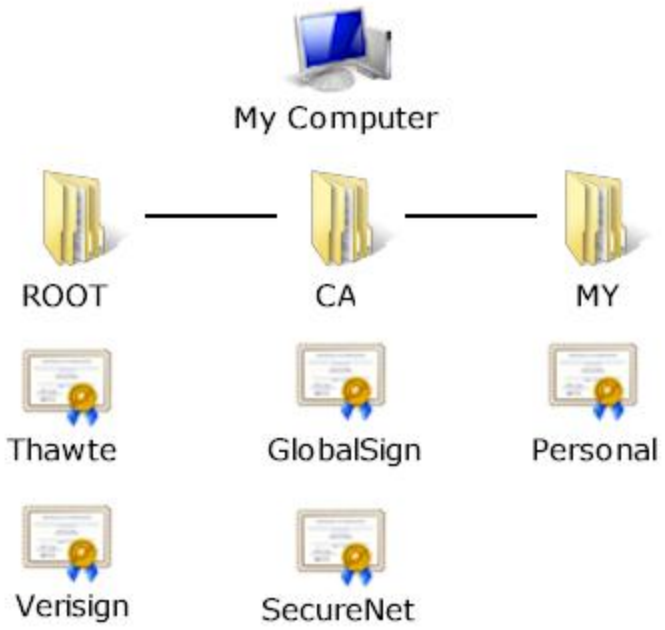
After the handshake phase has completed, the client may choose to examine the digital certificate that has been returned by the server. The information contained in the certificate includes the date that it was issued, the date that it expires, information about the organization who issued the certificate (called the issuer) and to whom the certificate was issued (called the subject of the certificate). The client may also validate the status of the certificate, determining if it was issued by a trusted certificate authority and was returned by the same company or individual it was issued to. There may be certain cases where the client determines that there's a problem with the certificate (for example, if the certificate's common name does not match the domain name of the server), but chooses to continue communicating with the server. Note that the connection with the server will still be secure in this case. In other cases, for example if the certificate has expired, the client may choose to terminate the connection and warn the user.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Digital Certificates

With secure connections, digital certificates are used to exchange public keys for data encryption and to provide identification information. This information typically includes the organization that was issued the certificate, its physical location and so on. The certificate itself is used to validate that the public key actually belongs to the entity it was issued to. The certificate also includes information about the Certification Authority (CA) who issued the certificate. The CA is responsible for validating the information provided by that organization, and then digitally signing the certificate. This establishes a relationship between the two so that when others validate the certificate, they know that it has been issued by a trusted third-party. For example, let's say that a company wants to implement a secure site so people can order products online. They would provide information about their company (organizational contacts, financial information and so on) to a trusted third party organization such as Verisign or DigiCert. That organization would then verify that the information they provided was complete and correct, and then would issue a signed certificate to them, which they install on their server. When a user connects to their server and checks the certificate, they see that it was issued by a trusted Certification Authority. In essence, the user is saying that because they trust the Certificate Authority, and the Certificate Authority trusts the company to whom the certificate was issued, they will trust the company as well.

To establish this relationship between the Certification Authority and the organization a certificate is issued to, there needs to be a root certificate which has been signed by the same trusted organization. This serves as the beginning of the certification path that is used to validate signed certificates. Using the above example, on the user's system there is a root certificate for Verisign, signed by Verisign. Root certificates are maintained in the local system's certificate store which is essentially a database of digital certificates. This database is structured so that different types of certificates can be organized in one central location on the system, and a standard interface is provided to enumerate and validate these certificates. Certificates are associated with a store name, allowing them to be easily categorized. For example, root certificates are stored under the name "Root", while a user's personal certificates (along with their private keys) are stored under the name "My".



When the Windows operating system is installed, there is a certificate store that contains the root certificates for the major Certification Authorities. However, there are situations where additional certificates may need to be added to the system. To facilitate this, there is a tool called CertMgr.exe which allows a user to install certificates, as well as export or remove certificates from the certificate store. When managing your system's certificate store, you should take the same care that you do when making changes to the system registry. Inadvertently removing a certificate could result in errors when attempting to access secure systems.

In general, the one situation where certificate management becomes important is when you want to develop your own secure server. This is because your server needs to have a signed certificate to send to the client in order to establish the secure connection. For general-purpose commercial applications, this generally means you would need to obtain a certificate that has been signed by a Certification Authority such as Verisign or DigiCert. This certificate would then be installed in the certificate store on the server. However, for development purposes it may be inconvenient to purchase a certificate. There also may be situations in which an organization wishes to function as its own Certification Authority and issue certificates themselves. This allows the organization to control how certificates are managed and can be ideal for secure applications that are designed for the corporate intranet. A utility for creating self-signed root certificates and server certificates is included with SocketTools.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Development

The SocketTools Library Edition provides a comprehensive collection of over eight hundred functions for performing a variety of Internet related programming tasks. Although the size of the SocketTools API may appear daunting, once you begin using the libraries in your own applications you'll find that the various libraries are designed to work together in a cohesive fashion. After you've familiarized yourself with one library, the others will become much simpler to use.

Throughout the Developer's Guide there are some general concepts and terminology used that are essential to understanding how SocketTools works. Each of these concepts is explored in detail, however a general, broad overview can also be useful when you are just getting started.

Protocols

A protocol, in terms of how the word is used in SocketTools, refers to the rules for how programs communicate with one another over a network. There are low level networking protocols such as TCP and UDP, as well as high level application protocols like FTP and HTTP. It can be helpful to think of a protocol as a sort of language; for two programs to communicate with each other, they must agree upon a protocol and understand how it is implemented.

Connections

The process of establishing a connection enables one program to communicate with another. Connection requests are made by client applications, and accepted by server applications. When the server accepts the connection request, the connection is completed. When you use the Connect function to successfully establish a connection to a server, a client session is created. SocketTools uses handles to reference specific sessions, and an application can create multiple client/server sessions if necessary.

Sessions

A session refers to an active connection between a client and server program. This term is typically used interchangeably with connection; however in some cases a single session may involve multiple network connections. For example, the File Transfer Protocol library establishes one connection, called the command channel, when the client initially connects to the server. However, when a file is being uploaded or downloaded, a second connection called the data channel is created just for that transfer. When the transfer completes, the second connection is terminated while the original command channel connection remains active. Even though there are multiple connections being made, SocketTools considers it to be a single client session. An active session is referenced by the handle which the Connect function returns to your program. When the session is no longer needed, the library's Disconnect function will terminate the connection to the server and release the resources allocated for that session. After that point, the handle is no longer valid and cannot be used in subsequent function calls.

Authentication

Many servers require that clients authenticate themselves by providing user names and passwords. Different application protocols implement several different types of authentication, and some protocols may support more than one authentication method. The SocketTools API provides one of two general types of authentication functions, depending on the protocol. For protocols which require the client to authenticate itself, the libraries will provide a Login function. Examples of this are **FtpLogin** and **ImapLogin**. For protocols where authentication is optional, the libraries will provide an Authenticate function. Examples of this are **HttpAuthenticate** and **SmtpAuthenticate**. Refer to the technical reference for the specific protocol to determine if authentication is required.

Events

Developers who use visual programming languages like Visual Basic will find the concept of events and event handling to be very familiar. In general terms, the SocketTools documentation uses "event" to refer to a mechanism where the library notifies the client that an operation has completed, some action has taken place or a change in status has occurred. SocketTools implements events by posting user-defined messages to the application and/or invoking callback functions that have been registered by the client. Events can be handled by the client either by processing the messages in the application's message queue, or by implementing callback functions and registering those functions with the library. One example of an event is a connection event, which is generated whenever an asynchronous network connection is completed by the client. Another example is a progress event, which is generated periodically by the library to inform the client of its progress as it sends or receives data. To determine what events are available in a specific library, refer to the documentation for its ***RegisterEvent*** function. More specific information about event handling is provided later in this guide.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Application Design

The SocketTools API is designed to be flexible enough to address the needs of developers who have very basic needs, as well as those who have more complex requirements. As a result, the functions for a library can be broken down into two general categories: a high level interface to perform common tasks, and a lower level interface which provides more control at the expense of being somewhat more complicated and requiring more coding. For example, consider the Hypertext Transfer Protocol (HTTP) library which has a variety of high level functions such as **HttpGetFile**, **HttpPostData** and so on. Using these functions, your application can perform the most common tasks for that protocol with a minimum of coding. You don't need to even understand the basics of how the protocol works, or what the library is doing. The high level functions allow you to program against the API as though it is a "black box", where you can provide the input and process the output without concerning yourself with the details of what's going on behind the scenes.

However, in some cases it's necessary for an application to have more direct control over how the library operates or to take advantage of features that aren't explicitly supported by one of the higher level APIs. As an example, the HTTP library also has functions like **HttpCommand** which enable you to send custom commands to a web server. Normally, for operations like retrieving a file or posting data to a script, this isn't necessary. But if your application needs to use WebDAV, a set of extensions to the HTTP protocol to support distributed web authoring, then the lower level functions like **HttpCommand** enable you to do this.

If you are generally new to Internet programming or are just getting started with the SocketTools Library Edition, we generally recommend that you begin familiarizing yourself with the higher level functions using a basic synchronous (blocking) connection in a single-threaded application. Once you become more familiar with how the library works, then you can move on to more complex applications which leverage the lower level API functions, taking advantage of multithreading, asynchronous networking connections and so on.

One of the common pitfalls that developers can encounter with a large toolkit like SocketTools is the inclination to over-design the application from the start, and then become frustrated because they don't yet have a clear picture of how all the pieces fit together. Start out with a basic design and then as you become more familiar with how the SocketTools libraries work, expand on it. Developers who are used to working with the Windows API will find themselves right at home, but even if you are new to Windows programming, you'll find that developing applications with SocketTools will soon become second nature.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Program Structure

Applications which use the SocketTools libraries will tend to have a similar structure, regardless of the specific protocol or programming language. While the details vary based on the library being used, the implementation can be broken down into several general steps:

- Initializing the library
- Connecting to the server
- Authenticating the client
- Performing one or more operations
- Disconnecting from the server
- Uninitializing the library

Initialization prepares the library to be used by your program, and is the first step that must be performed before you can use any other functions. Next, a connection is established with the server using the information provided by your program. For example, most of the connection functions require that you provide a host name, port number, a timeout period for synchronous operations and any additional options.

When the connection has completed, you will be given a handle to reference that client session; most of the SocketTools API require that you provide the function with a handle to identify the client session.

If the protocol requires that you authenticate the client in order to use the service, your application needs to provide this information. Once the client has been authenticated, it can then perform one or more operations, such as downloading a file, sending an email message and so on.

After you have finished, you disconnect from the server. Finally, before your program terminates, you uninitialize the library which causes it to perform any necessary housekeeping prior to releasing any system resources which were allocated on behalf of your program.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Library Initialization

When you begin developing your application using one of the SocketTools libraries, the first thing that you must do is initialize the library. This is done by calling the *Initialize* function for that library in your application's main thread. For example, the File Transfer Protocol library has a function named **FtpInitialize**, the Hypertext Transfer Protocol library has a function named **HttpInitialize** and so on. The one exception to this is the File Encoding library, which does not require initialization.

The initialization function serves two purposes. It loads the Windows networking libraries required to establish a connection and it validates the runtime license key that you provide. The runtime license key is a string of characters which identifies your license to use and redistribute the SocketTools libraries. It is unique to your product serial number and must be used when redistributing your application to an end-user.

Developers who are evaluating SocketTools will not have a runtime license key and must pass an empty string or a null pointer. This will enable the library to load on the development system during the evaluation period, but will prevent the library from being redistributed to an end-user until a license has been purchased.

If you install the product with a serial number, the runtime license key will be automatically created for you during the installation process. If you have installed an evaluation copy of SocketTools and then purchased a license, the license key can be created using the License Manager utility that was included with SocketTools. Simply select the **License** | **Header File** menu option and select the programming language that you are using. If your language is not listed, select Text File, which will create a simple text file with your license key.

The runtime license key is normally stored in the Include folder where you installed SocketTools and is defined in a file named "csrtkey12" which can be included with your application. For example, C/C++ programmers would use the **csrtkey12.h** header file while Visual Basic programmers would use the **csrtkey12.bas** module. The C++ header file would look something like this:

```
// SocketTools 12 Build 3024
// Copyright 2026 Catalyst Development Corporation
// All rights reserved
//
// This file is licensed to you pursuant to the terms of the
// product license agreement included with the original software
// and is protected by copyright law and international treaties.

#ifndef _INCLUDE_CSRTKEY12_H
#define _INCLUDE_CSRTKEY12_H

#ifndef UNICODE
#define CSTOOLS12_LICENSE_KEY ((LPCSTR)NULL)
#else
#define CSTOOLS12_LICENSE_KEY ((LPCWSTR)NULL)
#endif

#endif /* _INCLUDE_CSRTKEY12_H */
```

The macro CSTOOLS12_LICENSE_KEY specifies your runtime license key and this is what you would pass as the first argument to the initialization function. For example, here is how a C program would initialize the File Transfer Protocol library:

```
BOOL bInitialized;
bInitialized = FtpInitialize(CSTOOLS12_LICENSE_KEY, NULL);

if (!bInitialized)
{
    TCHAR szError[256];
    DWORD dwError;
```

```
    dwError = FtpGetLastError();
    FtpGetErrorString(dwError, szError, 256);
    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Note that in this case, the key is defined as NULL, which means that a runtime license key has not been created yet. The FtpInitialize function would succeed when the program was executed on the development system where the evaluation copy of SocketTools was installed, but would fail on another system. When a license is purchased and the runtime license key is generated, the csrtkey12.h header file would be replaced with the actual key value. After re-compiling your program, you would be able to redistribute the application to other systems.

C++ programmers who are using the class wrappers are not required to explicitly initialize the libraries because this is done automatically in the class constructor. Review the cstools12.h file which defines all of the SocketTools classes to see how the constructor initializes the class.

For another example of how you can initialize the libraries, consider the Visual Basic module **csrtkey12.bas** which defines the runtime license key:

```
'
' SocketTools 12 Build 3024
' Copyright 2026 Catalyst Development Corporation
' All rights reserved
'
' This file is licensed to you pursuant to the terms of the
' product license agreement included with the original software
' and is protected by copyright law and international treaties.
'
Public Const CSTOOLS12_LICENSE_KEY As String = ""
```

This is similar to the C/C++ header file and could either be included with your Visual Basic application or you could simply copy the string into your application. The equivalent to the C code listed above would look like this:

```
Dim bInitialized As Long
bInitialized = FtpInitialize(CSTOOLS12_LICENSE_KEY, 0)

If bInitialized = 0 Then
    Dim strError As String * 256
    Dim dwError As Long
    Dim nLength As Long
    dwError = FtpGetLastError()
    nLength = FtpGetErrorString(dwError, strError, 256)
    MsgBox Left(strError, nLength)
    Exit Sub
End If
```

In both examples, if the **FtpInitialize** function fails by returning a value of 0 (False), the program displays a message box to the user that explains the error and then exits.

An application is only required to call a library's initialization function once, but it must be called for each library that is used. If both the File Transfer Protocol and Hypertext Transfer Protocol libraries were being used in the same application, it would be required to call both **FtpInitialize** and **HttpInitialize** at the beginning of the program.

It is safe to call the initialization function more than once, but for each time that it is called, you must call the *Uninitialize* function for that library before your program terminates. For example, if you called **FtpInitialize** at the beginning of your program, you must call **FtpUninitialize** before your program ends. The *Uninitialize*

function performs any necessary housekeeping operations, such as returning memory allocated for the library back to the operating system. If there are any open connections at the time that the Uninitialize function is called, they will be aborted. After the library has been uninitialized, you must call the initialization function again in order to use any of the library's other functions.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Synchronous and Asynchronous Sockets

One of the first issues that you'll encounter when developing your application is the difference between synchronous (blocking) and asynchronous (non-blocking) connections. Whenever you perform some operation on a socket, it may not be able to complete immediately and return control back to your program. For example, a read on a socket cannot complete until some data has been sent by the server. If there is no data waiting to be read, one of two things can happen: the function can wait until some data has been written on the socket, or it can return immediately with an error that indicates that there is no data to be read.

The first case is called a synchronous or blocking socket. In other words, the program is "blocked" until the request for data has been satisfied. When the server does write some data on the socket, the read operation will complete and execution of the program will resume. The second case is called an asynchronous or non-blocking socket, and requires that the application recognize the error condition and handle the situation appropriately.

Programs that use asynchronous sockets typically use one of two methods when sending and receiving data. The first method is called polling and the program periodically attempts to read or write data from the socket, typically using a timer. The second method is to use what is called asynchronous event notification. This means that the program is notified whenever a socket event takes place, and in turn can respond to that event. For example, if the remote program writes some data to the socket, an event is generated so that program knows it can read the data from the socket at that point. Events can be in the form of Windows messages posted to the application's message queue, or as callback functions.

Synchronous Sockets

For historical reasons, the default behavior is for sockets to function synchronously and not return until the operation has completed. However, blocking sockets in Windows can introduce some special problems in single-threaded applications. To prevent the program from becoming non-responsive, the blocking function will enter what is called a "message loop" where it continues to process messages sent to it by Windows and other applications. Because messages are being processed, this means that the program can be re-entered at a different point with the blocked operation parked on the program's stack. For example, consider a program that attempts to read some data from the socket when a button is pressed. Because no data has been written yet, it blocks and the program goes into a message loop. The user then presses a different button, which causes code to be executed, which in turn attempts to read data from the socket, and so on.

To resolve the general problems with blocking sockets, the Windows Sockets standard states that there may only be one outstanding blocked call per thread of execution. This means that applications that are re-entered (as in the example above) will encounter errors whenever they try to take some action while a blocking function is already in progress. If the language supports the creation of threads, it is strongly recommended that the program create worker threads to perform any socket I/O.

There are significant advantages to using blocking sockets. In most cases, the application design and implementation is simpler, and raw throughput (the rate at which data is sent and received) is generally higher with blocking sockets because it does not have to go through an event mechanism to notify the application of a change in status. If you are using a programming language that supports multithreading, then the use of synchronous sockets is typically the best choice. However, if your are using an older language that does not provide support for multithreading, such as Visual Basic 6.0, and your program needs to establish multiple simultaneous connections, then an asynchronous, event-driven design is more appropriate.

Asynchronous Sockets

SocketTools facilitates the use of asynchronous sockets by generating events when appropriate. For example, an event occurs whenever the server writes on the socket, which tells your application that there is data waiting to be read.

In general, the use asynchronous sockets is preferred when you have a single-threaded application that must establish multiple, simultaneous connections with a server. In that situation, the use of non-blocking sockets avoids the restriction that prevents more than one outstanding socket operation in the thread and can enable the program to remain more responsive to the user. Practically speaking, there are few languages today that do not support multithreading, so this limitation tends to apply more to the legacy languages such as Visual Basic 6.0.

Best Practices

If your programming language of choice does support multithreading, it is recommended that you create worker threads to manage the sockets in your program. This leaves the main thread responsible for handling the user interface, and the worker threads can handle the network communications. There are some significant advantages to this approach:

- The networking code is generally isolated from the user interface, only requiring that the main UI thread be notified of the progress of the operation. For example, updating a progress bar control as the contents of a file is being downloaded. This tends to minimize any clutter in the UI code and creates a clear separation of functionality that will make the program easier to modify and maintain.
- Isolating the networking code in a worker thread ensures that there are no conflicts between other threads, including the main UI thread. Each thread effectively owns the sockets that it creates, and those sockets can be used independently of one another without concern about potential conflicts.
- Code written using synchronous sockets is typically easier to update, maintain and debug. The coding style lends itself to a more straight forward, top-down structure and logical errors are usually easier to find than with code written using asynchronous sockets.
- There is less overhead associated with synchronous sockets because no event mechanism is used, and handlers don't have to be implemented in callback functions. Event notifications that post messages to hidden window have to be processed through the message queue which is typically shared by the UI thread.
- Polling an asynchronous socket can cause spikes in CPU utilization and is generally not recommended. Applications which attempt to simulate blocking sockets by creating an asynchronous socket and then polling it can negatively impact the performance of the application, and in some cases the overall system.

In summary, there are three general approaches that can be taken when building an application with regard to blocking or non-blocking sockets:

- Use a synchronous (blocking) socket. In this mode, the program will not resume execution until the socket operation has completed. In a single-threaded application, blocking socket operations can cause code to be re-entered at a different point, leading to complex interactions (and difficult debugging) if there are multiple active connections in use by the application. If the programming language supports multithreading, it is recommended that each connection be isolated within its own worker thread.
- Use an asynchronous (non-blocking) socket, which allows your application to respond to events. For example, when the server writes data to the socket, an event is generated. Your application can respond by reading the data from the socket, and perhaps send some data back, depending on the context of the data received. The code required for managing asynchronous sockets can be more complex, however it is the best solution for single-threaded applications that must establish simultaneous connections.
- Use a combination of synchronous and asynchronous socket operations. The ability to switch between blocking and non-blocking modes "on the fly" provides a powerful and convenient way to perform socket operations under some circumstances. However, switching between blocking and non-blocking mode can make the application more complex and difficult to debug. It is important to note that the

warning regarding blocking sockets also applies here.

If you decide to use asynchronous sockets in your application, it's important to keep in mind that you must check the return value from every read and write operation. It is possible that your may not be able to send or receive all of the data specified at that time. Frequently, developers encounter problems when they write a program that assumes a given number of bytes can always be written to or read from the socket. In many cases, the program works as expected when developed and tested on a local area network, but fails unpredictably when the program is released to a user that has a slower network connection (such as a serial dial-up connection to the Internet). By always checking the return values of these operations, you ensure that your program will work correctly, regardless of the speed or configuration of the network.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Secure Connections

SocketTools supports secure network communications using both the Transport Layer Security (TLS) protocol and the Secure Shell (SSH) protocol. These technologies protect data from interception and tampering while it is transmitted across the network.

TLS is commonly used by Internet protocols such as HTTPS, FTPS, SMTP, POP3 and IMAP, while SSH is used for secure remote shell access, secure file transfer (SFTP) and secure copy (SCP) operations.

Technology	Common Protocols	Authentication
TLS	HTTPS, FTPS, IMAPS, POP3S, SMTPS	Server certificates, optional client certificates
SSH 2.0	SSH, SFTP, SCP	Passwords and public/private key pairs

Current versions of SocketTools support TLS 1.3 and TLS 1.2 by default and automatically negotiate the highest protocol version supported by both the client and server. Legacy versions of TLS, including TLS 1.0 and TLS 1.1, are also supported for compatibility with older systems, but they are disabled by default and must be explicitly enabled by the application.

Applications should use TLS 1.2 or TLS 1.3 whenever possible. Earlier protocol versions are provided only for compatibility with legacy systems and may not meet current security requirements.

SSH connections support password and public key authentication and can validate the server's host key to ensure that the client is communicating with the expected server. Unlike TLS, SSH does not use X.509 certificates and has its own authentication and key management mechanisms.

In most cases, enabling a secure connection is as simple as specifying the appropriate option when establishing a connection. SocketTools automatically negotiates the security settings required by the server and validates server certificates when using TLS. For example, when using the HTTP API, you would call the **HttpConnect** function with the `HTTP_OPTION_SECURE` option flag.

Implicit and Explicit TLS

Protocols that use TLS generally establish secure connections using either an implicit or explicit security model. SocketTools supports both approaches.

With an implicit secure connection, the client and server begin negotiating the TLS session immediately after the network connection is established. Servers that support implicit security typically listen on a dedicated port reserved for secure communications. For example, web servers normally accept standard HTTP connections on port 80 and secure HTTPS connections on port 443. When a client connects to port 443, the server automatically expects the connection to use TLS.

With an explicit secure connection, the client first establishes a standard connection and then requests that the session be upgraded to use TLS. For example, an FTP client may send the AUTH command to request a secure FTPS session. Protocols that support explicit security often use the same port for both secure and non-secure connections.

In most cases, SocketTools can automatically determine whether an explicit or implicit TLS session is required based on the protocol and port number selected. Your application can override the default behavior by setting options prior to establishing the connection.

TLS Client Certificates

Some servers require additional authentication through the use of a client certificate. In these cases, the application must provide a certificate before the secure connection is established. Client certificates may be used in addition to, or instead of, traditional username and password authentication.

For information about selecting and managing client certificates, refer to the documentation for the **SECURITYCREDENTIALS** structure in the Technical Reference.

SSH Authentication

SSH connections support both password authentication and public key authentication. Public key authentication uses a pair of cryptographic keys: a private key that is kept securely by the client and a corresponding public key that is installed on the server. During authentication, the server verifies that the client possesses the correct private key without requiring the key itself to be transmitted across the network.

Public key authentication is commonly used with SSH, SFTP and SCP servers because it provides stronger security than passwords and enables automated processes to connect without requiring interactive user input.

Public key authentication is supported by both the SSH and FTP APIs by providing the path to the private key file in the **SECURITYCREDENTIALS** structure before establishing the connection. When connecting to an SSH, SFTP or SCP server, the specified private key will be used for public key authentication if required by the server. SocketTools will automatically load the key and use it when authenticating with the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Input/Output

Each of the SocketTools networking libraries provides functions for sending and/or receiving data from a server. At the lowest level, this is done by calling the Write function for sending data and the Read function for receiving data. For example, the File Transfer Protocol library has the functions **FtpWrite** and **FtpRead**. In most cases, these functions exchange data as a stream of bytes without any regard for the actual content. For developers who are using Unicode, it is important to note that there are no Unicode versions of these functions. If the data being sent or received is textual, it is your program's responsibility to convert it into an appropriate format, typically using the **MultiByteToWideChar** and **WideCharToMultiByte** functions that are part of the standard Windows API.

When working at this very low level, it is important to understand how data is exchanged over the network. Many developers are inclined to think of the data that is sent or received in terms of discrete blocks, or packets. The expectation is that if they send a certain number of bytes of data in a single write, the server will receive that number of bytes in a single read. However, this is not how TCP works, and by extension, not how the SocketTools libraries work with regards to this kind of low level network I/O. The Transmission Control Protocol (TCP) is called a stream-oriented protocol because data is exchanged between the client and server as a stream of bytes. While TCP will guarantee that the data will arrive intact, with the bytes received in the same order that they were written, there is no guarantee that the amount of data received in a single read operation on the socket will match the amount of data sent by the remote host.

For example, consider a server that sends data to a client in four separate operations, each containing 1024 bytes of data. While it is convenient to think of these as discrete blocks of data, TCP considers it to be a stream of 4096 bytes. The client may receive that data in a single read on the socket, returning all 4096 bytes. Alternatively, it may read the socket, and only receive the first 1460 bytes; subsequent reads may return another 1460 bytes, followed by the remaining 1176 bytes. Applications which make assumptions about the amount of data they can read or write in a single operation may work in some environments, such as on a local network, but fail on slower connections.

A general rule to use when designing an application using TCP is to consider how the program would handle the situation where reading n bytes of data only returns a single byte. If the application can correctly handle this kind of extreme case, then it should function correctly even under adverse network conditions.

In some situations it may be desirable to design the application to exchange information as discrete messages or blocks of data. While this isn't directly supported by TCP, it can be implemented on top of the data stream. There are several methods that can be used to accomplish this, depending on the requirements of the application:

1. Exchange the data as fixed length structures. This is the simplest approach, and has very little or no overhead. The client and server can either use predefined values, or negotiate the size of the data structures when the connection is established.

2. Prefix variable-length data structures with the number of bytes being sent. The length value could be expressed either as a native integer value, or as a fixed-length string that is converted to a numeric value by the application. This allows the receiver to read this fixed length value, and then use that value to determine how many additional bytes must be read to obtain the complete message or data structure.

3. Prefix the data with a unique byte or byte sequence that would normally not be found in the data stream. This would be followed by the data itself, with a complete message received when another unique byte sequence is encountered. Alternatively, a unique byte sequence could be used to terminate a message. This is the approach that many Internet application protocols use, such as FTP, SMTP and POP3. Commands are sent as one or more printable characters,

terminated with a carriage-return (CR) and linefeed (LF) byte sequence that tells the server that a complete command has been received.

4. A combination of the above methods, using unique byte sequences, the message length and even additional information such as a CRC-32 checksum or MD5 hash to validate the integrity of the data. This would effectively create an "envelope" around the data being exchanged, and additional checks could be made to ensure that the data has been received and processed correctly.

Regardless of the method used, for best performance it is recommended that the application buffer the data received and then process the data out of that buffer. When using asynchronous (non-blocking) sockets, the application should read all of the data available on the socket, typically in a loop which adds the data to the buffer and exiting the loop when there is no more data available at that time.

It is important to keep in mind that all of this is only required if you decide to use the lower level APIs in the SocketTools libraries. The higher level APIs automatically handle the lower level network I/O for you. For example, the **FtpGetData** function will retrieve a file from the server and return the entire contents to your application in a single function call. When using the high level APIs, the details of how the data is read and processed is handled by the library and no additional coding is required on your part.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Event Handling

In SocketTools, event notification provides a mechanism for the library to inform the application of a change in the status of the current session. Events are generally divided into two general categories, asynchronous network events and status events.

Asynchronous network events occur when a non-blocking connection is established and a network event occurs, such as a connection completing or data arriving from the server. Status events are used to indicate a change in status, such as a blocking operation being canceled or the progress of an operation such as a file transfer.

Asynchronous network events require that the program provide a handle to a window that will receive the event message and a user-defined message identifier that will be used to identify the event to the application. Typically the window is an invisible window created specifically for the purpose of processing events, however it can also be a visible window used by the application such as a form or dialog. If the window handle refers to a visible window, it is important to remember that if the window is destroyed, asynchronous event notification will automatically be disabled. In general it is recommended that the application use a private, invisible window if asynchronous network events are required. To enable event handling, each of the networking libraries has a function named ***EnableEvents*** which accepts a handle to a window and a unique message identified. There is also a ***DisableEvents*** function which will disable event handling. For example, the Hypertext Transfer Protocol library has functions called **HttpEnableEvents** and **HttpDisableEvents**. Note that if you initially establish an asynchronous connection, it is not necessary to explicitly call the ***EnableEvents*** function for the library since you will already have a non-blocking connection and events will be posted to the specified window.

In addition to posting event notifications to a window, it is also possible to register an event handler that will be called by the library whenever the event occurs. In this case, you would use the ***RegisterEvent*** function for the library, where you would specify the event that you wish to register, along with a user-defined parameter that will be passed to the callback function.

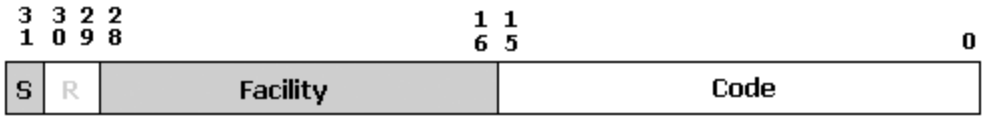
In some cases, it is desirable to suspend event handling for a session. To accomplish this, the libraries have a function called ***FreezeEvents*** which can be used to suspend and resume event handling. While events are frozen, network events will be queued rather than posted to the notification window. When event handling is resumed, those events will be fired and event handling will resume normally. It is recommended that applications freeze event handling when they are performing some action where it would not be desirable for the event handler to be executed. Examples of where this would be appropriate would be a modal dialog or a message box, where the user must respond before the application continues executing.

An important consideration when it comes to event handling is that all asynchronous network events are level triggered. This means that once an event is fired, it will not be fired again until some action is taken by the application to handle the event. This is most commonly found with events that are generated when the server sends data to your application, signaling to you that there is data available to be read. Even though the server may continue sending you more data, another event will not be generated until you read at least some of the data that has been sent to you. This is done to prevent the application from being flooded with event notifications. However, failure to handle an event can cause event notification to appear to stall. If you determine that you cannot handle an event at the time, you must freeze event handling. You can resume event notification when you are in a position to process events again.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Error Handling

Typically an error is indicated when a SocketTools function either returns a value of zero (false) or -1. When an error occurs in one of the libraries, the *GetLastError* function can be used to obtain the error code. The *GetLastError* function returns a unsigned 32-bit integer value that is unique to each thread in a given application. Each library also has it's own private function which will return the last error code for the current thread. For example, the File Transfer Protocol library has a function called **FtpGetLastError**. SocketTools error codes conform to the standard format used by Windows:



The **S** bitfield is used to refer to the severity of the error. A value of zero specifies success, while a value of one specifies an error condition.

The **R** bitfield is reserved for future use by Windows, and must be set to zero. Applications should not take any action based on the values of these two bits.

The **Facility** bitfield specifies which group of status codes the error belongs to. The SocketTools libraries use FACILITY_ITF, which is designated for use by third-party libraries.

The **Code** bitfield specifies the actual error code. This may correspond to a Windows Sockets error, or an error that is unique to the SocketTools libraries.

Each of the SocketTools error codes has a matching constant, defined in the header files or modules provided for your programming language. These constants should always be used for comparison against the value returned by the *GetLastError* function.

For each error code, there is a matching description of the error which can be used by the application. Applications may choose to use these error descriptions when displaying a message box to a user or for internal logging purposes. To retrieve the description of the error, the **FormatMessage** function can be used. Each library also has its own private function to return an error description which may be easier to use. For example, the File Transfer Protocol library has a function called **FtpGetErrorString**, the Hypertext Transfer Protocol library has a function called **HttpGetErrorString** and so on.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Debugging Facilities

All of the SocketTools networking libraries include a built-in facility for generating debugging output in the form of a log file that provides information about the internal functions that it is using and the data that is being exchanged between the client and server. This is commonly referred to in the documentation as generating a trace log or enabling function logging.

To create a log file, your application must then call the ***EnableTrace*** function in the library. For example, if you were using the Simple Mail Transfer Protocol library, you would call the function **SmtpEnableTrace**. This function requires two arguments: the name of a file that the trace log data will be written to and a numeric value which specifies the level of the trace. The default level, zero, specifies that general information about the function calls being made will be logged. The most detailed logging is provided by specifying a level of four. In that case, all data exchanged between your application and the server is logged. This provides the most information, however it also generates the largest log files. To disable logging, use the ***DisableTrace*** function for the library, such as **SmtpDisableTrace**.

There are two important things that you need to consider when enabling trace logging. The first is that the log file is always appended to, never overwritten by the library. This means that the files can grow to be very large, particularly with trace that includes all of the data sent and received by your application. You can use the standard file I/O functions in your language to manage the log file or even write your own data out to the file. Each time the file is written to, SocketTools will open the file, append the logging data and then close the file. The libraries will never keep the file open between operations. This is important because if your application terminates abnormally, it ensures all of the logging data has been written and there are no open file handles being held by one of the libraries. However, this does incur additional overhead and can impact the performance of your application. When possible, it is recommended that you enable logging around the functions that you feel may be part of the problem you're trying to resolve, and then disable logging when it is no longer required. Simply enabling logging at the beginning of your application can result in unnecessarily large log files.

If your application uses multiple SocketTools libraries, it is only necessary to enable logging in one of them. Once enabled, all SocketTools network operations in the current thread will be logged, regardless of which library was used to enable logging.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Language Support

The SocketTools Library Edition can be used with a wide variety of programming languages and software development tools for Windows. The majority of the library functions use simple data types such as 16-bit and 32-bit integers, and null character terminated strings. To determine if your development language is capable of using the Library Edition, it should support all of the following features:

- It needs the ability to load a dynamic-link library (DLL) and call exported functions from that library. In some cases, as with C or C++, you can use an import library instead of dynamically loading the library. This allows you to link your program just as you would a standard function library. If the language does not support the use of import libraries, it must provide a mechanism for declaring a function and specifying the arguments that will be passed to it.
- Languages must call the SocketTools functions using the **stdcall** calling convention. Parameters are pushed on to the stack in reverse order (from right to left), and the called function is responsible for clearing the stack. Note that this is different from the standard C/C++ calling convention in which the stack is cleared by the calling function, and from the Pascal calling convention in which arguments are pushed on the stack left to right.
- The language must support basic integer data types, including 16-bit, 32-bit and 64-bit integers. The language must also support a string data type that is represented as an array of bytes, terminated by a null character (a character with the ASCII value of zero). If a different native string type is used, the language must provide a means to convert between the native string format and a null-terminated byte array.
- The language must support passing function parameters by value and by reference. When a variable is "passed by value", a copy of its value is passed to the function on the stack. However, when a variable is "passed by reference", the memory address of the variable (typically called a pointer) is passed to the function. In most cases, the functions in the SocketTools library expect integer data to be passed by value, while string and data structures are passed by reference.
- In addition to passing a variable by reference to a function, the language must provide the ability to allocate a block of memory of arbitrary size and be able to pass its address to a function. For example, in C there are the malloc() and free() functions, and in Visual Basic there are the Dim and ReDim statements.
- The language must support calling functions which do not return a value. For example, in C and C++, such functions are declared as void. In Visual Basic, a function which does not return a value is declared as Sub (a subroutine).

Although not required, it is recommended that the language also support the ability to create user-defined data structures. For example, in C and C++, the struct keyword is used to define a data structure. In Visual Basic, the Type statement is used to create user-defined data structures.

Microsoft Visual C++, Visual C#, Visual Basic.NET, Visual Basic 6.0, C++ Builder and Delphi are all examples of languages which can use the SocketTools Library Edition. If your programming language is capable of calling native Windows API functions, then you should be able to use SocketTools. Consult your language technical reference for additional information about how to call functions exported from a standard Windows dynamic link library.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Data Types

Because various languages handle data types in different ways, the SocketTools libraries have been designed to primarily use basic data types such as integers and strings. The following is a list of numeric data types that are used, along with their C++, C# and Visual Basic equivalents.

Description	Size	Range	Win32 API	C++	C#	VB 6	VB.NET
Byte	1 byte	0 to 255	BYTE	uint8_t	byte	Byte	Byte
Boolean	4 bytes	0 is False, 1 or -1 is True	BOOL	int32_t	int	Long	Integer
Handle	4 bytes	0 to 4,294,967,295	HCLIENT	void *	IntPtr	Long	IntPtr
Integer	4 bytes	-2,147,483,648 to 2,147,483,647	INT	int32_t	int	Long	Integer
Integer	4 bytes	0 to 4,294,967,295	UINT	uint32_t	uint	Long	UInteger
Short Integer	2 bytes	-32,768 to 32,767	SHORT	int16_t	short	Integer	Short
Short Integer	2 bytes	0 to 65,535	WORD	uint16_t	ushort	Integer	UShort
Long Integer (32-bit)	4 bytes	-2,147,483,648 to 2,147,483,647	LONG	int32_t	int	Long	Integer
Long Integer (32-bit)	4 bytes	0 to 4,294,967,295	DWORD	uint32_t	uint	Long	UInteger
Long Integer (64-bit)	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	LONGLONG	int64_t	long	N/A	Long
Long Integer (64-bit)	8 bytes	0 to 18,446,744,073,709,551,615	ULONGLONG	uint64_t	ulong	N/A	ULong

One problem that is frequently encountered when converting function definitions from C or C++ to other languages is the size of the integer data type. For example, default integer size for Visual Basic 6 is 16-bits on 32-bit platforms. However, in Visual Basic .NET, as well as languages like Visual C++, the default integer size is 32-bits. Also, some languages do not support unsigned integer types. In this case, as with Visual Basic, the signed type should be used instead.

Visual Basic 6.0 does not have a native 64-bit integer type, however a 64-bit value can be represented with a structure. Most SocketTools APIs which use 64-bit integers either represent them as LARGE_INTEGER or LONGLONG values. In Visual Basic, the structure would look like this:

```
Public Type LARGE_INTEGER
    LoPart As Long
    HiPart As Long
End Type
```

This would contain the lower 32-bit and upper 32-bit values of the 64-bit integer. This same structure is also used in various Windows API functions. The same approach can also be used with other languages which do not natively support 64-bit integers, but are capable of passing small structures by value as arguments to functions.

Boolean Data

Boolean parameters present a special problem for two reasons. Firstly, the data types used to represent boolean values frequently vary between languages. Secondly, different languages represent the values "true" and "false" differently. boolean parameters (declared as BOOL in the function prototypes) should always be passed as 32-bit signed integers.

It is important to note that in C++ the **bool** type is an unsigned 8-bit integer and is different than BOOL, which is a signed 32-bit integer used with SocketTools functions. The .NET languages also have System.boolean which is not a numeric value, but rather a structure type where the only values are true or false. The SocketTools function definitions for Visual Basic and C# use marshaling to change the numeric values used by the SocketTools API to native .NET boolean types.

If you are passing a boolean parameter to a function, then "false" should be represented by a value of zero and "true" as a non-zero value, typically a value of one. When writing code that checks a boolean flag, or tests a boolean return value from a function, it is recommended that you test against a value of zero. For example, consider the following code in Visual Basic 6.0:

```
Dim bConnected As Long
bConnected = InetIsConnected(hSocket)
If bConnected = True Then
    ' The socket is connected to a server
Else
    ' The socket is not connected, or the socket handle
    ' may be invalid; call InetGetLastError to check the
    ' error code
End If
```

In this example, even if the function **InetIsConnected** was successful, the program would always believe that it failed because of the explicit test against the value True. This is because the function returns a value of 1 to indicate success, but Visual Basic defines True as -1. Instead, the code should be written as:

```
Dim bConnected As Long
bConnected = InetIsConnected(hSocket)
If bConnected <> 0 Then
    ' The socket is connected to a server
Else
    ' The socket is not connected, or the socket handle
    ' may be invalid; call InetGetLastError to check the
    ' error code
End If
```

In summary, the rule of thumb for dealing with boolean parameters is that they should always be 32-bit integer values, and you should always compare the boolean against a value of zero, never against a predefined constant like True.

String Data

String parameters can also present a challenge when calling functions from languages other than C and C++. Different languages tend to have different internal representations of how string data is stored. The convention used by the SocketTools libraries is that a string is an array of characters, terminated by a null character (a character with an ASCII value of zero). The length of the string is not stored in the string data itself, and by definition, a string cannot contain embedded nulls.

The size of a character depends on whether the ANSI or Unicode version of a function is being called. SocketTools includes support for both types of strings and which you use generally depends on how you have your project configured. For example, consider the function **InetReadLine** which reads a line of text from a socket and returns

it in a string buffer. There are actually two versions of this function, **InetReadLineA** which uses an 8-bit ANSI string buffer, and **InetReadLineW** which accepts a 16-bit Unicode string buffer.

Depending on the language, **InetReadLine** may be a macro which expands to the appropriate version based on the project. This is the case for C/C++ projects, where which version of a function is called is determined by the project settings and whether or not the UNICODE macro is defined. For other languages, one version of the function will always be used. For example, with Visual Basic 6.0 only the ANSI versions of the functions are defined, while for C# and Visual Basic .NET only the Unicode version of functions are declared, with marshaling used to convert them to the native System.String type.

To use functions which require string parameters, the language must be capable of converting between its native string data type and the null-terminated character array expected by the SocketTools functions. For example, Object Pascal provides the **StrPCopy** function. Note that Visual Basic provides implicit conversion between its native string type and null-terminated strings when a string is passed by value (ByVal) instead of by reference.

If you are unsure of how your language handles null-terminated strings, we recommend that you review the language's technical reference for information on how to call native Windows API functions. Since the Windows API also uses null-terminated strings, that same information can be used to determine how to call functions in the SocketTools libraries.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Unicode

Unicode is an international character standard that assigns a unique code point to every character used in modern computing. Unlike traditional character sets, Unicode is not limited by region or language. On Windows, Unicode text is typically represented using UTF-16 encoding, where each character is encoded as one or two 16-bit values.

The SocketTools libraries support both UTF-8 and UTF-16 by providing two versions of each function that either expects a string as an argument (including those functions which pass structures that contain strings) or returns the address of a string. The functions which expect ASCII and/or UTF-8 encoding have a suffix of "A", while the functions which use Unicode strings have a suffix of "W" (wide). No suffix is used with functions which expect binary (non-textual) data or only use numeric parameters and return numeric values.

For example, consider the **InetGetLocalName** function. If you looked at the list of exported functions in the library, you would see two functions exported, **InetGetLocalNameA** and **InetGetLocalNameW**. In C and C++, that function that is called depends on how the application is being built. That is, if your application is built to use Unicode (in other words, the UNICODE macro is defined and you are linking with Unicode versions of the standard libraries), then the **InetGetLocalNameW** function will be used instead of **InetGetLocalNameA**. In other languages, you may have to explicitly declare which version of the function you wish to use. For example, in Visual Basic the Alias keyword must be used with the function declaration to specify the correct name.

It is important to understand that most functions which end with "A" in the SocketTools API expect input strings to contain either ASCII text or valid UTF-8 encoded text. Localized ANSI code pages are not reliably compatible with most Internet protocols because they can result in incomplete or inconsistent character conversions between systems using different code pages. This can lead to corrupted text or unexpected behavior when processing internationalized content.

Recent versions of Windows also provide an option to use UTF-8 as the active system code page for non-Unicode applications. While this can improve compatibility with internationalized text in ANSI applications, legacy software may not always function correctly when this option is enabled. It is recommended that modern applications use native UTF-16 Unicode support whenever possible, and this is the default configuration for modern Windows development tools such as Visual Studio.

Character Encoding and Unicode

While Unicode defines a vast character set, it must be encoded to store or transmit strings using those characters. The most common encodings are:

- **UTF-8** is a variable-length encoding where ASCII characters use 1 byte, and other characters use 2-4 bytes. This is the standard for the Internet.
- **UTF-16** is used internally by Windows APIs. Characters are typically 2 bytes, with some requiring surrogate pairs.
- **UTF-32** is a fixed-width 4-byte encoding, rarely used in practice. This is not used with the Windows API.

Automatic Conversion

When building a project configured to use the Unicode character set, SocketTools will automatically convert strings to UTF-8 encoded text before transmitting that data over the network. This conversion only occurs with string types, and will not be performed on byte arrays or other types of data that is not represented as a null-terminated string value.

Converting strings to UTF-8 encoding ensures textual data is exchanged in a uniform way which is not affected by the local system's localization and language settings. Virtually all modern servers on the Internet today expect text to be exchanged using UTF-8 encoding, and because ASCII characters are considered a subset of

UTF-8, they are not subject to conversion.

Earlier versions of SocketTools always performed Unicode string conversions using the default system code page, rather than using UTF-8 encoding. This change will not typically affect most applications; however, if you are using Unicode strings, it is important to keep in mind that this conversion to UTF-8 can change how data is exchanged over the network. If you want to prevent this automatic UTF-8 encoding, you can perform the preferred conversion in your code (for example, using the **WideCharToMultiByte** function) and then explicitly call the ANSI version of the SocketTools function. It is important to note that most ANSI functions in SocketTools expect UTF-8 as the default encoding and it is recommended that modern applications use Unicode whenever possible.

The Encoding and Compression library includes helper functions that can simplify the process of performing UTF-8 encoding and decoding. The **IsUnicodeText** function will analyze a string buffer to determine if it contains valid Unicode text. The **UnicodeDecodeText** and **UnicodeEncodeText** functions can be used to perform conversions between UTF-8 encoded text and Unicode strings.

Strings and Byte Arrays

Some SocketTools functions require you to use byte arrays instead of strings, regardless of the character set your project uses. This can create problems when reading and writing Unicode string data. For example, consider the **InetRead** and **InetWrite** functions which are used to read and write data on a socket. Because character strings and byte arrays are essentially identical when using an ANSI character set, a C/C++ programmer may try to write code such as this:

```
LPTSTR lpszData = _T("This is a test, this is only a test");
INT cchData = lstrlen(lpszData);
INT nResult;

nResult = InetWrite(hSocket, lpszData, cchData);
```

This would work as expected until you change your project to use the Unicode character set. The problem is that the Unicode string is no longer an array of 8-bit bytes, but is now an array of 16-bit integers. The Unicode string must be converted to a byte array before passing it to the **InetWrite** function. One way to do this is to use the **WideCharToMultiByte** function:

```
LPTSTR lpszData _T("This is a test, this is only a test");
INT nLength = lstrlen(lpszData);

#ifdef UNICODE
// If the project is built using Unicode then we should convert
// the string to UTF-8 before writing it to the socket
LPBYTE lpBuffer = (LPBYTE)malloc((nLength + 1) * 4);

if (lpBuffer == NULL)
{
    // Unable to allocate memory for conversion
    return;
}

nLength = WideCharToMultiByte(CP_UTF8, 0,
                              (LPCWSTR)lpszData,
                              cchData,
                              (LPSTR)lpBuffer,
                              ((nLength + 1) * 4),
                              NULL, NULL);

if (nLength == 0)
{
```

```
    // Unable to convert the string to UTF-8
    free(lpBuffer);
    return;
}
#else
lpBuffer = (LPBYTE)lpszData;
#endif

INT nBytesWritten = InetWrite(hSocket, lpBuffer, nLength);

#ifdef UNICODE
// Make sure we free the memory used for the conversion
free(lpBuffer);
#endif
```

The type of characters being converted may also present a problem to the developer. In this example, the string is easily converted because it is composed only of characters that are part of the basic ASCII character set. In most cases it is recommended that you use CP_UTF8 to convert the text to UTF-8. When converting a string that contains international characters, such as accented vowels, the conversion using the system code page may result in unprintable characters. For additional information, check your programming language's technical reference for issues with regards to localization and the use of Unicode.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Microsoft Visual C++

The SocketTools Library Edition includes import libraries, header files and class wrappers which can be used with Microsoft Visual C++ 10 and later. If you are using Visual Studio 2010, you should have Service Pack 1 installed. If you are using Visual Studio 2012 or 2013, you should have Update 5 installed. If you are using Visual Studio 2015, you should have Update 3 installed. For Visual Studio 2017 and later, use the Visual Studio Installer to ensure that you have the latest updates installed.

To build an application using SocketTools with Visual C++, the first step is to make sure the compiler is configured correctly to specify the correct paths to the directories where the header files and import libraries are found. In Visual C++, this is done by selecting Tools | Options from the menu, and then selecting the Directories tab. In Visual Studio, this is done by selecting Tools | Options from the menu, then selecting the Projects folder and the VC++ Directories properties page. The path for the include files should be the Include folder, and the path for the library files should be the Lib folder, found where you installed SocketTools.

When you install SocketTools, an environment variable named **SocketTools12** will be created which provides the full path to the installation folder. In Visual Studio and other IDEs, you can typically reference this environment variable rather than hard-coding the path. For example, the include folder would be **$(SocketTools12)\Include**. For compatibility with both 32-bit and 64-bit Windows platforms, SocketTools will be installed in the **C:\Program Files (x86)\SocketTools 11.0 Library Edition** folder on 64-bit Windows, and **C:\Program Files\SocketTools 12.0 Library Edition** on 32-bit Windows.

If you are programming in C or C++, you must include the header file **cstools12.h** in your program. This file contains all of the macro, type, structure and function prototype definitions used by the SocketTools libraries. This will also automatically include two other header files which are used by SocketTools. The **cserror12.h** header file defines all the error codes, and the **csrtkey12.h** header defines the runtime license key which is used to initialize the SocketTools libraries.

When you include the SocketTools header file, it will automatically include **windows.h** and several other standard Windows API header files. If you include the Windows header file before you include the SocketTools header, you will get an error indicating that you must include the SocketTools header file first. This is because the Windows header file will attempt to include the (now deprecated) **winsock.h** header by default. If possible, include the SocketTools header file first and there won't be a need to explicitly include the Windows header. Alternatively, you can define the macro **WIN32_LEAN_AND_MEAN** which tells the Windows header to only include other essential header files. Your application should never explicitly include either **winsock.h** or **winsock2.h** because those are automatically included by the SocketTools header.

If you are using Microsoft Foundation Classes (MFC) with your project, you will need to include **afx.h** before you include the SocketTools header file. Also make sure your project has explicitly defined the **_WIN32_WINNT** macro to indicate the minimum target platform for your project. If it is undefined, a warning will be issued and it will default to the highest operating system version supported by the Windows development kit you have installed.

A collection of classes which encapsulate the SocketTools API are also included as part of the standard header file. The SocketTools classes can be used with or without the Microsoft Foundation Classes (MFC), although certain functions are only available if MFC is used. The classes themselves are designed to be basic wrappers around the API and it is expected that most developers will use them as the base class for their own classes.

For Visual C++ programmers, there are special macros which are used in the **cstools12.h** header file which controls certain behavior:

CSTOOLS_NO_LIBRARIES
Visual C++ supports a pragma which can be used to automatically specify the names of libraries that the compiler should attempt to link with in order to resolve function calls. By default, it is not necessary to explicitly specify the names of the SocketTools import libraries to link to. However, if you wish to

explicitly link to specific import libraries, then define this macro prior to including the cstools12.h header file.

CSTOOLS_NO_NAMESPACE
When compiling a C++ application, the SocketTools functions are defined in a namespace called SocketTools. This prevents the possibility of conflicts with functions of the same name that may be used in other libraries. If you prefer the functions to be defined in the global namespace instead, then define this macro prior to including the cstools12.h header file.

CSTOOLS_NO_CLASSES
When compiling a C++ application, class wrappers for the SocketTools API are normally included with the function prototypes. Defining this macro prevents the C++ classes from being defined, limiting the application to using only the standard functions.

These macros are not normally defined, and in most cases they will not be needed. Please note that if for some reason you modify the header files directly, you should copy the changes to another file using a different name. Otherwise, when you install an update to the product, the **cstools12.h** and **cserror12.h** header files will be replaced and you will lose your modifications.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Microsoft Visual C#

The SocketTools Library Edition provides a C# module in the Include folder named **cstools12.cs** which can be included with your projects. This defines a namespace called SocketTools which contains classes for each of the libraries, as well as the constants and error codes. It also defines some standard Windows API functions which can be helpful when using some of the SocketTools functions.

Only the Unicode versions of the SocketTools API functions are defined for C# applications. The .NET Framework uses Unicode strings as its native string type, and the runtime automatically marshals string values between managed and unmanaged code. Because Unicode support is built into the framework, separate ANSI declarations are not necessary. Applications should always use the Unicode function definitions when calling the SocketTools Library Edition APIs from managed code.

String Arguments

Although the SocketTools libraries are unmanaged code, for the most part you won't have to be concerned about marshaling data. However, there are a few special considerations that programmers should be aware of when using the SocketTools API. The first deals with string buffers that are passed by reference and modified by the function. An example of this would be the **HttpGetErrorString** function, which is used to obtain a description of a specific error. In C/C++ you would allocate a character array and pass it to the function, such as:

```
TCHAR szError[256];
DWORD dwError;
INT nLength;

dwError = HttpGetLastError();
nLength = HttpGetErrorString(dwError, szError, 256);
MessageBox(NULL, szError, "Error", MB_OK | MB_ICONEXCLAMATION);
```

In C#, the HttpGetErrorString function declaration uses the **StringBuilder** class as the string argument. The equivalent code in C# would be:

```
StringBuilder strError = new StringBuilder(256);
uint dwError;
int nLength;

dwError = HttpGetLastError();
nLength = HttpGetErrorString(dwError, strError, 256);
MessageBox.Show(strError.ToString(), "Error",
                MessageBoxButtons.OK,
                MessageBoxIcon.Exclamation);
```

Make certain you use the **System.Text** namespace which defines the **StringBuilder** class. Also keep in mind that if you want to pass a **StringBuilder** variable to a function that expects a **String**, you should use the **ToString** method.

Byte Array Arguments

A number of SocketTools functions use byte arrays, either as an input argument to the function, or as an output argument which will contain data when the function returns. An example of this is the **HttpGetData** function, which will access a resource on the server and return the contents of that resource in a byte array passed to the function. For example, the following code in C++ would return the first 1024 bytes of the index page on a web server:

```
BYTE byteBuffer[1024];
DWORD dwLength;
INT nResult;
```

```
dwLength = sizeof(byteBuffer);
nResult = HttpGetData(hClient,
                      "/index.html",
                      byteBuffer,
                      &dwLength,
                      0);
```

If the function is successful, the byteBuffer array will contain the first 1024 bytes of the index page. In C#, the equivalent code would look like this:

```
byte[] byteBuffer = new byte[1024];
uint dwLength;
int nResult;

dwLength = (uint)byteBuffer.Length;
nResult = HttpGetData(hClient,
                      "/index.html",
                      byteBuffer,
                      ref dwLength,
                      0);
```

In C++, byte arrays can be used interchangeably with ANSI strings. However, in C# you will need to use the **System.Text.ASCIIEncoding** class to convert a byte array into a string. For example:

```
String strBuffer = (new ASCIIEncoding()).GetString(byteBuffer, 0, (int)dwLength);
```

This would convert the contents of the byte array into a String. Note that you should only do this if the data returned by the function is actually text. In this example, it is acceptable to do because the byte array contains the HTML text for the index page.

Global Memory Handles

In addition to using byte arrays, some SocketTools functions can use global memory handles (HGLOBALs) to exchange large amounts of data. Using the Windows API, global memory handles are allocated by the **GlobalAlloc** function, dereferenced by the **GlobalLock** function and released by the **GlobalFree** function. These handles can be used in C# by using the **System.Runtime.InteropServices** marshaling classes. There are also some helper functions defined in the SocketTools.Win32 class which can be used to dereference and release global memory handles.

An application may choose to use a global memory handle instead of a pre-allocated buffer if the amount of data is very large, or the total amount of data that will be returned is unknown at the time the function is being called. Consider the call to the **HttpGetData** function used in the previous example. A pre-allocated buffer of 1024 bytes was passed to the function, and it copied up to that amount of data into the buffer. However, what if you wanted the complete page and did not know how large it was? You could attempt to determine the size of the page that was being requested using the **HttpGetFileSize** function, and then use that value to allocate a buffer. However, this incurs additional overhead and it is not always possible to get the size of a resource on a web server. Another alternative would be to simply allocate a very large buffer, but this could result in the application allocating large amounts of memory that it doesn't use and you would still run the risk that it wouldn't be large enough.

The solution for this problem is to use a global memory handle rather than a pre-allocated buffer. Instead of copying the data into a buffer, the function allocates a global memory handle and stores the contents in the memory referenced by that handle. When the function returns, it passes the handle back to the caller. The caller then dereferences the handle to access the memory, and releases the handle when it is no longer needed. Here is an example of how it would be used in C/C++:

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
INT nResult;
```

```
nResult = HttpGetData(hClient,
                      "/index.html",
                      &hgblBuffer,
                      &dwLength,
                      0);

if (nResult != HTTP_ERROR)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Do something with the data and then unlock and
    // release the handle when it is no longer needed

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Note that the global memory handle is initialized to NULL and the length argument is initialized to zero. This is important to do because this is how the function knows that it should be returning a global memory handle instead of copying data into a buffer. If you forget to initialize those arguments, the function will fail and may cause the application to terminate with a general protection fault.

The equivalent code in C# would look like this:

```
uint hgblBuffer = 0;
uint dwLength = 0;
int nResult;

nResult = HttpGetData(hClient,
                      "/index.html",
                      ref hgblBuffer,
                      ref dwLength,
                      0);

if (nResult != HTTP_ERROR)
{
    IntPtr lpBuffer = Win32.GlobalLock(hgblBuffer);
    String strBuffer = Marshal.PtrToStringUTF8(lpBuffer);

    // Do something with the data and then unlock and
    // release the handle when it is no longer needed

    Win32.GlobalUnlock(hgblBuffer);
    Win32.GlobalFree(hgblBuffer);
}
```

Because the global buffer is a byte array internally, you would normally convert it from a UTF-8 string. Older versions of .NET may not support the **PtrToStringUTF8** method, and instead you would need to use **PtrToStringAnsi**. However, be aware that if the data is UTF-8 encoded, it may not be converted correctly. In this case, you may wish to copy the data to a byte array and perform the conversion in your application.

It is important to remember to unlock and release the global memory handle when you are no longer using it. Those handles are not managed by the Common Language Runtime (CLR) garbage collector, so if you forget to release them, the application will have a memory leak.

Because you are dealing directly with memory buffers, the normal safety checks performed by C# are not available, such as making sure you are not exceeding the bounds of an array. It is recommended that you always test your code carefully and always save your current project before debugging or executing the program.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read. An event handler is simply a callback function which has a specific set of arguments, and the address of that function is passed to the **RegisterEvent** function in the library. However, C# doesn't permit you to simply pass the address of a function in the same way that you can in C++. Instead, you need to use what are called delegates. In .NET, a delegate is a reference type that is used to encapsulate a method that has a specific set of arguments, known as its signature. Although delegates can seem confusing at first, it is easiest to think of them as function pointers. They're basically used in the same way that function pointers are used in C++, except that they're type-safe.

The first step to creating an event handler is to create a method in your class which matches the signature of the callback function defined by the SocketTools event notification function. When the event handler is called, SocketTools will pass the handle to the client session, an event identifier to specify which event occurred, an error code value if an error occurred, and the user-defined value that was specified by the caller when the event was registered. In C++, the callback function would be declared as:

```
VOID EventHandler(HCLIENT hClient,
                  UINT nEventId,
                  DWORD dwError,
                  DWORD_PTR dwParam)
```

In C#, the equivalent method would be defined inside the class as:

```
private void EventHandler(IntPtr hClient,
                          uint nEventId,
                          uint dwError,
                          IntPtr dwParam)
```

Let's create an event handler that updates a progress bar as a file is being downloaded from an HTTP server. To do this, we'll create a method called httpEventHandler:

```
private void httpEventHandler(IntPtr hClient, uint nEventId, uint dwError, IntPtr
dwParam)
{
    switch (nEventId)
    {
        case HTTP_EVENT_PROGRESS:
       {
           HTTPTRANSFERSTATUS httpStatus = new HTTPTRANSFERSTATUS();
           HttpGetTransferStatus(hClient, ref httpStatus);

           progressBar1.Value = (int)((double)(100.0 *
               (double)httpStatus.dwBytesCopied /
               (double)httpStatus.dwBytesTotal));
       }
       break;
    }
}
```

This event handler checks to see if the event ID indicates that it is a progress event, and if it is, creates a HTTPTRANSFERSTATUS structure and calls **HttpGetTransferStatus** to determine the number of bytes which have been copied so far. This is used to calculate a percentage and a progress bar control is updated with that value.

Now that the event handler has been written, the next step is to create the event delegate for the method. For each of the SocketTools networking libraries that support event notification, there is a delegate defined, such as **HttpEventDelegate**. The delegate is created using code like this:

```
HttpEventDelegate httpEventProc = new HttpEventDelegate(httpEventHandler);
```

The httpEventProc variable is now like a function pointer which can be passed to **HttpRegisterEvent** in order to enable notification for that event:

```
int nResult = HttpRegisterEvent(
                hClient,
                HTTP_EVENT_PROGRESS,
                httpEventProc,
                0);
```

The first argument to **HttpRegisterEvent** is the handle to the client session. The second argument is the event ID for which you want to enable notification, the third argument is the event delegate, and the fourth argument is a user-defined value. That same value is passed to the event handler as the dwParam argument.

Your event handler is now registered, and SocketTools will call your event handler during the process of downloading or uploading a file to notify you of the progress of the transfer.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Microsoft Visual Basic .NET

The SocketTools Library Edition provides a Visual Basic .NET module in the Include folder named **cstools12.vb** which can be included with your projects. This defines the constants and functions in the SocketTools libraries.

Only the Unicode versions of the SocketTools API functions are defined for VB .NET applications. The .NET Framework uses Unicode strings as its native string type, and the runtime automatically marshals string values between managed and unmanaged code. Because Unicode support is built into the framework, separate ANSI declarations are not necessary. Applications should always use the Unicode function definitions when calling the SocketTools Library Edition APIs from managed code.

String Arguments

An important consideration when using the SocketTools libraries in Visual Basic is how string arguments are being used by the function. In most cases, the string is provided as input to the function, such as the hostname or address of a server to establish a connection with. However, in some cases the string is passed to the function as an output buffer into which the function copies data. For example, the **InetGetLocalName** function stores the local host name into a string parameter. A Visual Basic programmer may write code that looks like this:

```
Dim strLocalName As String
Dim nLength As Integer

nLength = InetGetLocalName(strLocalName, 256)
```

Although this code looks correct, it will invariably result in a general protection fault or some other unpredictable error. The problem is that although the ***strLocalName*** variable has been defined, no memory has been allocated for it. To do this, you need to declare the string as:

```
Dim strLocalName As String = New String(Chr(0), 256)
Dim nLength As Long

nLength = InetGetLocalName(strLocalName, 256)
```

This will create a string that is filled with null characters. However, when the function returns the string, it will be padded with those null characters and they should be removed. The complete example would be written in Visual Basic as:

```
Dim strLocalName As String = New String(Chr(0), 256)
Dim nLength As Long

nLength = InetGetLocalName(strLocalName, 256)
strLocalName = Strings.Left(strLocalName, nLength)
```

Because the function returns the length of the string, the Left method can be used to truncate the string using that length. However, if the function does not return the string length, you'll need to use the TrimEnd method to trim the string at the terminating null character, such as:

```
' Trim string up to the terminating null character
strLocalName = strLocalName.TrimEnd(vbNullChar.ToCharArray())
```

The first method is more efficient, but requires that the function return the number of characters it copied into the string. The second method is slightly less efficient, but will work even if the function does not return the string length.

Byte Array Arguments

A number of SocketTools functions use byte arrays, either as an input argument to the function, or as an output argument which will contain data when the function returns. An example of this is the **HttpGetData** function, which will access a resource on the server and return the contents of that resource in a byte array

passed to the function. For example, the following code in C++ would return the first 1024 bytes of the index page on a webserver:

```
BYTE byteBuffer[1024];
DWORD dwLength;
INT nResult;

dwLength = sizeof(byteBuffer);
nResult = HttpGetData(hClient,
                      "/index.html",
                      byteBuffer,
                      &dwLength,
                      0);
```

If the function is successful, the byteBuffer array will contain the first 1024 bytes of the index page. In Visual Basic.NET, the equivalent code would look like this:

```
Dim byteBuffer(1024) As Byte
Dim dwLength As Long
Dim nResult As Long

dwLength = UBound(byteBuffer)
nResult = HttpGetData(hClient, _
                      "/index.html", _
                      byteBuffer(0), _
                      dwLength, _
                      0)
```

In C++, byte arrays can be used interchangeably with ANSI strings. However, in Visual Basic.NET you will need to use the **System.Text.Encoding** class to convert a byte array into a string. For example:

```
Dim strBuffer As String
Dim Encoding As System.Text.Encoding = _
    System.Text.Encoding.GetEncoding(1252)

strBuffer = Encoding.GetString(byteBuffer).Substring(0, wwLength)
```

This would convert the contents of the byte array into a String. Note that you should only do this if the data returned by the function is actually text. In this example, it is acceptable to do because the byte array contains the HTML text for the index page. Note that the encoding is also explicitly set to code page 1252 (ISO Latin 1), which ensures that any characters with the high bit set are converted correctly.

Global Memory Handles

In addition to using byte arrays, some SocketTools functions can use global memory handles (HGLOBALs) to exchange large amounts of data. Using the Windows API, global memory handles are allocated by the **GlobalAlloc** function, dereferenced by the **GlobalLock** function and released by the **GlobalFree** function. These handles can be used in Visual Basic.NET with the helper functions defined in the SocketTools module.

An application may choose to use a global memory handle instead of a pre-allocated buffer if the amount of data is very large, or the total amount of data that will be returned is unknown at the time the function is being called. Consider the call to the **HttpGetData** function used in the previous example. A pre-allocated buffer of 1024 bytes was passed to the function, and it copied up to that amount of data into the buffer. However, what if you wanted the complete page and did not know how large it was? You could attempt to determine the size of the page that was being requested using the **HttpGetFileSize** function, and then use that value to allocate a buffer. However, this incurs additional overhead and it is not always possible to get the size of a resource on a web server. Another alternative would be to simply allocate a very large buffer, but this could result in the application allocating large amounts of memory that it doesn't use and you would still run the risk that it wouldn't be large enough.

The solution for this problem is to use a global memory handle rather than a pre-allocated buffer. Instead of copying the data into a buffer, the function allocates a global memory handle and stores the contents in the memory referenced by that handle. When the function returns, it passes the handle back to the caller. The caller then dereferences the handle to access the memory, and releases the handle when it is no longer needed. Here is an example of how it would be used in C/C++:

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
INT nResult;

nResult = HttpGetData(hClient,
                      "/index.html",
                      &hgblBuffer,
                      &dwLength,
                      0);

if (nResult != HTTP_ERROR)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Do something with the data and then unlock and
    // release the handle when it is no longer needed

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Note that the global memory handle is initialized to NULL and the length argument is initialized to zero. This is important to do because this is how the function knows that it should be returning a global memory handle instead of copying data into a buffer. If you forget to initialize those arguments, the function will fail and may cause the application to terminate with a general protection fault.

The equivalent code in Visual Basic.NET would look like this:

```
Dim hgblBuffer As Integer = 0
Dim dwLength As Integer = 0
Dim nResult As Long

nResult = HttpGetData(hClient,
                      "/index.html",
                      hgblBuffer,
                      dwLength,
                      0);

If nResult <> HTTP_ERROR Then
    Dim lpBuffer As IntPtr
    Dim byteBuffer() As Byte
    ReDim byteBuffer(dwLength - 1)

    ' Lock the global memory handle and copy the
    ' contents of the buffer into the byte array
    lpBuffer = GlobalLock(hgblBuffer)
    CopyMemory(byteBuffer(0), lpBuffer, dwLength)

    ' Unlock and release the global memory handle
    GlobalUnlock(hgblBuffer)
    GlobalFree(hgblBuffer)

    ' Do something with the data
```

```
End If
```

If you wanted to convert the contents of the global memory buffer into a string, the Marshal class has a helper function which enables you to do this easily. To use it, you should import **System.Runtime.InteropServices** and then use the **PtrToStringUTF8** method. For example:

```
If nResult <> HTTP_ERROR Then
    Dim lpBuffer As IntPtr
    Dim strBuffer As String

    ' Lock the global memory handle and use the
    ' PtrToStringAnsi function to return a string
    lpBuffer = GlobalLock(hgblBuffer)
    strBuffer = Marshal.PtrToStringUTF8(lpBuffer)

    ' Unlock and release the global memory handle
    GlobalUnlock(hgblBuffer)
    GlobalFree(hgblBuffer)

    ' Do something with the data
End If
```

Because the global buffer is a byte array internally, you would normally convert it from a UTF-8 string. Older versions of .NET may not support this method, and instead you would need to use **PtrToStringAnsi**. However, be aware that if the data is UTF-8 encoded, it may not be converted correctly. In this case, you may wish to copy the data to a byte array and perform the conversion in your application.

It is important to remember to unlock and release the global memory handle when you are no longer using it. Those handles are not managed by the Common Language Runtime (CLR) garbage collector, so if you forget to release them, the application will have a memory leak.

Because you are dealing directly with memory buffers, the normal safety checks performed by Visual Basic are not available, such as making sure you are not exceeding the bounds of an array. It is recommended that you always test your code carefully and always save your current project before debugging or executing the program.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or progress notifications during a file transfer. An event handler is simply a callback function which has a specific set of arguments, and the address of that function is passed to the **HttpRegisterEvent** function in the library. However, Visual Basic .NET doesn't permit you to simply pass the address of a function in the same way that you can in C++ or Visual Basic 6.0. Instead, you need to use what are called delegates. In .NET, a delegate is a reference type that is used to encapsulate a method that has a specific set of arguments, known as its signature. Although delegates can seem confusing at first, it is easiest to think of them as function pointers. They're basically used in the same way that function pointers are used in C++, except that they're type-safe.

The first step to creating an event handler is to create a method in your class which matches the signature of the callback function defined by the SocketTools event notification function. When the event handler is called, SocketTools will pass the handle to the client session, an event identifier to specify which event occurred, an error code value if an error occurred, and the user-defined value that was specified by the caller when the event was registered. In C++, the callback function would be declared as:

```
VOID EventHandler(HCLIENT hClient,
                  UINT nEventId,
                  DWORD dwError,
                  DWORD_PTR dwParam)
```

In Visual Basic.NET, the equivalent method would be defined inside the form or class as:

```
Private Sub EventHandler(ByVal hClient As IntPtr, _
                         ByVal nEventId As Integer, _
                         ByVal dwError As Integer, _
                         ByVal dwParam As IntPtr)
```

Let's create an event handler that updates a progress bar as a file is being downloaded from an HTTP server. To do this, we'll create a method called **HttpEventHandler**:

```
Private Sub HttpEventHandler(ByVal hClient As IntPtr, _
                             ByVal nEventId As Integer, _
                             ByVal dwError As Integer, _
                             ByVal dwParam As IntPtr)
    Select Case nEventId
    Case HTTP_EVENT_PROGRESS
        Dim httpStatus As HTTPTRANSFERSTATUS
        HttpGetTransferStatus(hClient, httpStatus)
        ProgressBar1.Value = CInt(100.0# * _
            CDbl(httpStatus.dwBytesCopied) / _
            CDbl(httpStatus.dwBytesTotal))
    End Select
End Sub
```

This event handler checks to see if the event ID indicates that it is a progress event, and if it is, calls **HttpGetTransferStatus** to determine the number of bytes that have been copied so far. This is used to calculate a percentage and a progress bar control is updated with that value.

Now that the event handler has been written, the next step is to create the event delegate for the method. For each of the SocketTools networking libraries that support event notification, there is a delegate type defined. The delegate is created using code like this:

```
Dim httpEventProc As HttpEventDelegate = New HttpEventDelegate(AddressOf
HttpEventHandler)
```

The httpEventProc variable is now like a function pointer which can be passed to HttpRegisterEvent in order to enable notification for that event:

```
nResult = HttpRegisterEvent( _
             hClient, _
             HTTP_EVENT_PROGRESS, _
             httpEventProc, _
             0)
```

The first argument to **HttpRegisterEvent** is the handle to the client session. The second argument is the event ID for which you want to enable notification, the third argument is the event delegate, and the fourth argument is a user-defined value. That same value is passed to the event handler as the dwParam argument.

Your event handler is now registered, and SocketTools will call your event handler during the process of downloading or uploading a file to notify you of the progress of the transfer.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Microsoft Visual Basic 6.0

The SocketTools Library Edition provides a module for Visual Basic 6.0 in the Include folder named **cstools12.bas** which can be included with your projects. This defines the constants and functions in the SocketTools libraries. It is recommended that you install at least Visual Studio 6.0 Service Pack 6 (SP6) for Visual Basic 6.0.

Note that because this module is very large, Visual Basic may report an error when attempting to compile a program that uses it. This is a limitation of Visual Basic and doesn't indicate a problem with the module itself. To resolve the problem, you can copy the declarations that you need into your own private module and use those instead.

An alternative to using the function declarations in a module is to create a reference to the library in your project. Although the SocketTools libraries are not ActiveX DLLs, they do contain a type library resource which defines the constants and functions for that specific library. To create a reference to the library, first open your current project or create a new project. Then select the Project | References menu option. This will display a dialog box that lists the currently selected and available references. Click on the browse button and choose the appropriate library from the Windows system directory. The following libraries can be referenced:

Library Name	Library Description
csdnsv12.dll	Domain Name Service
csftpv12.dll	File Transfer Protocol
cshtpv12.dll	Hypertext Transfer Protocol
csicmv12.dll	Internet Control Message Protocol
csmapv12.dll	Internet Message Access Protocol
csmsgv12.dll	Mail Message
csmtpv12.dll	Simple Mail Transfer Protocol
csncdv12.dll	File Encoding Library
csnvtv12.dll	Terminal Emulation
csnwsv12.dll	Network News Transfer Protocol
cspopv12.dll	Post Office Protocol
csrshv12.dll	Remote Command Protocol
csrssv12.dll	News Feed Library
cstimv12.dll	Network Time Protocol
cstntv12.dll	Telnet Protocol
cstxtv12.dll	Text Message Library
cswhov12.dll	Whois Protocol
cswskv12.dll	SocketWrench (Windows Sockets)

Note that if you have the SocketTools ActiveX Edition installed, there may be references to the SocketTools controls included in the list. You do not want to select these since they refer to the ActiveX controls, not the libraries. Once the library has been referenced, you will be able to use the functions and constants detailed in the technical reference.

String Arguments

An important consideration when using the SocketTools libraries in Visual Basic is how string arguments are being used by the function. In most cases, the string is provided as input to the function, such as the hostname or address of a server to establish a connection with. However, in some cases the string is passed to the function as an output buffer into which the function copies data. For example, the **InetGetLocalName** function stores the local host name into a string parameter. A Visual Basic programmer may write code that looks like this:

```
Dim strLocalName As String
Dim nLength As Long

nLength = InetGetLocalName(strLocalName, 256)
```

Although this code looks correct, it will invariably result in a general protection fault or some other unpredictable error. The problem is that although the ***strLocalName*** variable has been defined, no memory has been allocated for it. There are two ways this can be done in Visual Basic. One is to declare the string as fixed-length, such as:

```
Dim strLocalName As String * 256
Dim nLength As Long

nLength = InetGetLocalName(strLocalName, 256)
```

The other is to dynamically allocate memory for the string using the **String** function, such as:

```
Dim strLocalName As String
Dim nLength As Long

strLocalname = String(256, 0)
nLength = InetGetLocalName(strLocalName, 256)
```

One final consideration is that string data returned by the function will be null-terminated. Because Visual Basic strings are managed differently, we need to remove the trailing null characters. The complete example would be written in Visual Basic as:

```
Dim strLocalName As String
Dim nLength As Long

strLocalname = String(256, 0)
nLength = InetGetLocalName(strLocalName, 256)
strLocalName = Left(strLocalName, nLength)
```

Because the function returns the length of the string, the Left function can be used to truncate the string using that length. However, if the function does not return the string length, you'll need to search for the terminating null character and trim the string from that position, such as:

```
' Trim string up to the terminating null character
strLocalName = Left(strLocalName, InStr(strLocalName, Chr(0))-1)
```

The first method is more efficient, but requires that the function return the number of characters it copied into the string. The second method is slightly less efficient, but will work even if the function does not return the string length.

Byte Array Arguments

A number of SocketTools functions use byte arrays, either as an input argument to the function, or as an output argument which will contain data when the function returns. An example of this is the **HttpGetData** function, which will access a resource on the server and return the contents of that resource in a byte array passed to the function. For example, the following code in C++ would return the first 1024 bytes of the index page on a webserver:

```
BYTE byteBuffer[1024];
DWORD dwLength;
INT nResult;

dwLength = sizeof(byteBuffer);
nResult = HttpGetData(hClient,
                      "/index.html",
                      byteBuffer,
                      &dwLength,
                      0);
```

If the function is successful, the byteBuffer array will contain the first 1024 bytes of the index page. In Visual Basic 6.0, the equivalent code would look like this:

```
Dim byteBuffer(1024) As Byte
Dim dwLength As Long
Dim nResult As Long

dwLength = UBound(byteBuffer)
nResult = HttpGetData(hClient, _
                      "/index.html", _
                      byteBuffer(0), _
                      dwLength, _
                      0)
```

In C++, byte arrays can be used interchangeably with ANSI strings. However, in Visual Basic you will need to use the **StrConv** function to convert a byte array into a string. For example:

```
Dim strBuffer As String
strBuffer = Left(StrConv(byteBuffer, vbUnicode), dwLength)
```

This would convert the contents of the byte array into a String. Note that you should only do this if the data returned by the function is actually text. In this example, it is acceptable to do because the byte array contains the HTML text for the index page.

Global Memory Handles

In addition to using byte arrays, some SocketTools functions can use global memory handles (HGLOBALs) to exchange large amounts of data. Using the Windows API, global memory handles are allocated by the **GlobalAlloc** function, dereferenced by the **GlobalLock** function and released by the **GlobalFree** function. These handles can be used in Visual Basic with the helper functions defined in the SocketTools module.

An application may choose to use a global memory handle instead of a pre-allocated buffer if the amount of data is very large, or the total amount of data that will be returned is unknown at the time the function is being called. Consider the call to the **HttpGetData** function used in the previous example. A pre-allocated buffer of 1024 bytes was passed to the function, and it copied up to that amount of data into the buffer. However, what if you wanted the complete page and did not know how large it was? You could attempt to determine the size of the page that was being requested using the **HttpGetFileSize** function, and then use that value to allocate a buffer. However, this incurs additional overhead and it is not always possible to get the size of a resource on a web server. Another alternative would be to simply allocate a very large buffer, but this could result in the application allocating large amounts of memory that it doesn't use and you would still run the risk that it wouldn't be large enough.

The solution for this problem is to use a global memory handle rather than a pre-allocated buffer. Instead of copying the data into a buffer, the function allocates a global memory handle and stores the contents in the memory referenced by that handle. When the function returns, it passes the handle back to the caller. The caller then dereferences the handle to access the memory, and releases the handle when it is no longer needed. Here is an example of how it would be used in C/C++:

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
INT nResult;

nResult = HttpGetData(hClient,
                      "/index.html",
                      &hgblBuffer,
                      &dwLength,
                      0);

if (nResult != HTTP_ERROR)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Do something with the data and then unlock and
    // release the handle when it is no longer needed

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Note that the global memory handle is initialized to NULL and the length argument is initialized to zero. This is important to do because this is how the function knows that it should be returning a global memory handle instead of copying data into a buffer. If you forget to initialize those arguments, the function will fail and may cause the application to terminate with a general protection fault.

To work with the global memory handles used by SocketTools, you will need to define a few standard functions that are part of the Windows operating system:

```
Declare Function GlobalLock Lib "kernel32.dll" ( _
    ByVal hMem As Long _
    ) As Long

Declare Function GlobalUnlock Lib "kernel32.dll" ( _
    ByVal hMem As Long _
    ) As Long

Declare Function GlobalFree Lib "kernel32.dll" ( _
    ByVal hMem As Long _
    ) As Long

Declare Sub CopyMemory Lib "kernel32.dll" _
    Alias "RtlMoveMemory" ( _
    ByRef lpDestination As Byte, _
    ByVal lpSource As Long, _
    ByVal dwLength As Long)
```

The **GlobalLock** function is used to dereference a global memory handle, returning the address in memory where the data is stored. The **GlobalUnlock** function is used to unlock a memory handle that was previously locked with a call to **GlobalLock**. The **GlobalFree** function releases the memory back to the operating system. The **CopyMemory** function is used to copy the data into a byte array that your program has allocated.

Because the function declaration for the **HttpGetData** function expects you to pass a byte array, you will need to redefine the function in a private module. Note that this declaration will override how the function is declared if you have referenced the library. The standard function declaration for **HttpGetData** looks like this:

```
Declare Function HttpGetData Lib "cshtpv12.dll" _
    Alias "HttpGetDataA" ( _
    ByVal hClient As Long, _
    ByVal lpszResource As String, _
```

```
    ByRef lpBuffer As Byte, _
    ByRef lpdwLength As Long, _
    ByVal dwOptions As Long _
    ) As Long
```

The new declaration should be modified so that the third argument is changed to a Long passed by reference:

```
Declare Function HttpGetData Lib "cshtpv12.dll" _
    Alias "HttpGetDataA" ( _
    ByVal hClient As Long, _
    ByVal lpszResource As String, _
    ByRef hgblBuffer As Long, _
    ByRef lpdwLength As Long, _
    ByVal dwOptions As Long _
    ) As Long
```

That long integer will contain the global memory handle allocated by the function when it returns. With these changes, the equivalent code in Visual Basic 6.0 would look like this:

```
Dim hgblBuffer As Long
Dim dwLength As Long
Dim nResult As Long

hgblBuffer = 0
dwLength = 0

nResult = HttpGetData(hClient,
                      "/index.html",
                      hgblBuffer,
                      dwLength,
                      0);

If nResult <> HTTP_ERROR Then
    Dim lpBuffer As Long
    Dim byteBuffer() As Byte
    ReDim byteBuffer(dwLength - 1)

    lpBuffer = GlobalLock(hgblBuffer)
    CopyMemory byteBuffer(0), lpBuffer, dwLength

    Dim strBuffer As String
    strBuffer = StrConv(byteBuffer, vbUnicode)

    GlobalUnlock hgblBuffer
    GlobalFree hgblBuffer
End If
```

It is important to remember to unlock and release the global memory handle when you are no longer using it. If you forget to release them, the application will have a memory leak. Because you are dealing directly with memory buffers, the normal safety checks performed by Visual Basic are not available, such as making sure you are not exceeding the bounds of an array. Simple mistakes such as passing an incorrect argument or the wrong buffer size can result in Visual Basic becoming unstable or terminating with a general protection fault. It is recommended that you always test your code carefully and always save your current project before debugging or executing the program.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or progress notifications during a file transfer. An event handler is simply a callback function which has a specific set of arguments, and the address of that function is passed to the

HttpRegisterEvent function in the library. In Visual Basic 6.0, you can use the **AddressOf** operator to obtain the address of a public callback function to pass to the function.

The first step to creating an event handler is to create a function which matches the signature of the callback function defined by the SocketTools event notification function. Your event handler must be created in a module, not a class or form, and you must declare it as public. When the event handler is called, SocketTools will pass the handle to the client session, an event identifier to specify which event occurred, an error code value if an error occurred, and the user-defined value that was specified by the caller when the event was registered. In C++, the callback function would be declared as:

```
VOID EventHandler(HCLIENT hClient,
                  UINT nEventId,
                  DWORD dwError,
                  DWORD_PTR dwParam)
```

In Visual Basic, the equivalent method would be defined as:

```
Public Sub HttpEventHandler(ByVal hClient As Long, _
                            ByVal nEventId As Long, _
                            ByVal dwError As Long, _
                            ByVal dwParam As Long)
```

Let's create an event handler that updates a progress bar as a file is being downloaded from an HTTP server. To do this, we'll create a function called **HttpEventHandler**:

```
Public Sub HttpEventHandler(ByVal hClient As Long, _
                            ByVal nEventId As Long, _
                            ByVal dwError As Long, _
                            ByVal dwParam As Long)
    Select Case nEventId
    Case HTTP_EVENT_PROGRESS
        Dim httpStatus As HTTPTRANSFERSTATUS
        HttpGetTransferStatus hClient, httpStatus
        Form1.ProgressBar1.Value = CInt(100# * _
            CDbl(httpStatus.dwBytesCopied) / _
            CDbl(httpStatus.dwBytesTotal))
    End Select
End Sub
```

This event handler checks to see if the event ID indicates that it is a progress event, and if it is, calls **HttpGetTransferStatus** to determine the number of bytes which have been copied so far. This is used to calculate a percentage and a progress bar control is updated with that value.

Now that the event handler has been written, the next step is to register the event handler with the library:

```
nResult = HttpRegisterEvent( _
            hClient, _
            HTTP_EVENT_PROGRESS, _
            AddressOf HttpEventHandler, _
            0)
```

The first argument to **HttpRegisterEvent** is the handle to the client session. The second argument is the event ID that you want to enable notification for, the third argument is the address of the event handler, and the fourth argument is a user-defined value. That same value is passed to the event handler as the dwParam argument.

Your event handler is now registered, and SocketTools will call your event handler during the process of downloading or uploading a file to notify you of the progress of the transfer.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Object Pascal and Delphi

The SocketTools Library Edition provides an Object Pascal unit in the Include folder named **cstools12.pas** which can be included with your projects.

Handles

Most of the functions in SocketTools use handles to reference client or server connections. In the C API, these handles are opaque unsigned integer values defined as `UINT_PTR` and there are typedefs which define types like `HCLIENT` and `HSERVER` as handle values. For 32-bit applications, the handles values are 32-bits wide and for 64-bit applications, the handles are 64-bits wide. The value `INVALID_HANDLE` is defined as `((UINT_PTR)-1)` which is the same as `INVALID_HANDLE_VALUE` defined in the Win32 API.

In Pascal, handles are represented by the `THandle` type. This is equivalent to the `UINT_PTR` type in C, and `INVALID_HANDLE` is defined as `THandle(-1)`. When you review a C prototype in the documentation and see a parameter type like `HCLIENT`, you want to use a `THandle` type.

String Arguments

An important consideration when using the SocketTools libraries in Delphi is how string arguments are being used by the function. In most cases, the string is provided as input to the function, such as the hostname or address of a server to establish a connection with. However, in some cases the string is passed to the function as an output buffer into which the function copies data. An example of this would be the **HttpGetErrorString** function, which is used to obtain a description of a specific error. In C/C++ you would allocate a character array and pass it to the function, such as:

```
VOID ShowError()
{
  TCHAR szError[256];
  DWORD dwError;

  dwError = HttpGetLastError();
  if (dwError > 0)
  {
      HttpGetErrorString(dwError, szError, 256);
      MessageBox(NULL, szError, "Error", MB_OK);
  }
}
```

Delphi has a type called **PAnsiChar** which is a pointer to a null terminated array of characters. It was created primarily as a compatibility type to work with the Windows API and C/C++ dynamic link libraries, and this is the type that the SocketTools functions use to represent strings. The equivalent code in Delphi would be:

```
procedure ShowError;
var
  dwError: LongWord;
  szError: Array [0..255] Of AnsiChar;
begin
  dwError := HttpGetLastError();
  if dwError > 0 then;
  begin
    HttpGetErrorString(dwError, szError, 256);
    ShowMessage(szError);
  end;
end;
```

Early versions of Delphi had helper functions such as **StrNew** to help convert and allocate null-terminated strings. However, those functions have been deprecated and in most cases current versions of Delphi will automatically convert between character arrays and its native **String** type. There are also a number of

functions available which are designed specifically to work with null-terminated strings. Refer to your language reference for more information about how Delphi uses null-terminated strings.

Byte Array Arguments

A number of SocketTools functions use byte arrays, either as an input argument to the function, or as an output argument which will contain data when the function returns. An example of this is the HttpGetData function, which will access a resource on the server and return the contents of that resource in a byte array passed to the function. For example, the following code in C++ would return the first 1024 bytes of the index page on a web server:

```
BYTE byteBuffer[1024];
DWORD dwLength;
INT nResult;

dwLength = sizeof(byteBuffer);
nResult = HttpGetData(hClient,
                      "/index.html",
                      byteBuffer,
                      &dwLength,
                      0);
```

If the function is successful, the byteBuffer array will contain the first 1024 bytes of the index page. In Delphi, the equivalent code would look like this:

```
var
  nResult: Integer;
  byteBuffer: Array [0..1023] Of Byte;
  dwLength: LongWord;
begin
  dwLength := 1024;
  nResult := HttpGetData(hClient,
                         PAnsiChar('/index.html'),
                         @byteBuffer,
                         dwLength,
                         0);
end;
```

In this example, you will notice that the string was explicitly cast to a null-terminated string using PAnsiChar. This is necessary because the **HttpGetData** function is overloaded and Delphi will complain that the function cannot be found if the cast is omitted.

If you wanted to convert the data in the byte array to a String type, you might think that you could do something like this:

```
strBuffer := PAnsiChar(@byteBuffer);
```

This may appear to work, but it would depend on there being a null character to terminate the byte array and it could result in garbage characters appearing at the end of your string or cause the program to attempt to read memory that it has not allocated. To convert the byte array into a string, use the following:

```
SetLength(strBuffer, dwLength);
Move(byteBuffer, strBuffer[1], dwLength);
```

Note that you should only do this if the data returned by the function is actually text. In this example, it is acceptable to do because the byte array contains the HTML text for the index page.

Global Memory Handles

In addition to using byte arrays, some SocketTools functions can use global memory handles (HGLOBALs) to exchange large amounts of data. Using the Windows API, global memory handles are allocated by the **GlobalAlloc** function, dereferenced by the **GlobalLock** function and released by the **GlobalFree** function.

These handles can be used in Delphi with the helper functions defined in the SocketTools unit.

An application may choose to use a global memory handle instead of a pre-allocated buffer if the amount of data is very large, or the total amount of data that will be returned is unknown at the time the function is being called. Consider the call to the **HttpGetData** function used in the previous example. A pre-allocated buffer of 1024 bytes was passed to the function, and it copied up to that amount of data into the buffer. However, what if you wanted the complete page and did not know how large it was? You could attempt to determine the size of the page that was being requested using the **HttpGetFileSize** function, and then use that value to allocate a buffer. However, this incurs additional overhead and it is not always possible to get the size of a resource on a web server. Another alternative would be to simply allocate a very large buffer, but this could result in the application allocating large amounts of memory that it doesn't use and you would still run the risk that it wouldn't be large enough.

The solution for this problem is to use a global memory handle rather than a pre-allocated buffer. Instead of copying the data into a buffer, the function allocates a global memory handle and stores the contents in the memory referenced by that handle. When the function returns, it passes the handle back to the caller. The caller then dereferences the handle to access the memory, and releases the handle when it is no longer needed. Here is an example of how it would be used in C/C++:

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
INT nResult;

nResult = HttpGetData(hClient,
                      "/index.html",
                      &hgblBuffer,
                      &dwLength,
                      0);

if (nResult != HTTP_ERROR)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Do something with the data and then unlock and
    // release the handle when it is no longer needed

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Note that the global memory handle is initialized to NULL and the length argument is initialized to zero. This is important to do because this is how the function knows that it should be returning a global memory handle instead of copying data into a buffer. If you forget to initialize those arguments, the function will fail and may cause the application to terminate with a general protection fault.

The equivalent code in Delphi would look like this:

```
var
  nResult: Integer;
  hgblBuffer: LongWord;
  dwLength: LongWord;
  pszBuffer: PAnsiChar;
begin
  hgblBuffer := 0;
  dwLength := 0;

  nResult := HttpGetData(hClient,
                         PChar('/index.html'),
```

```
                          hgblBuffer,
                          dwLength,
                          0);

    if nResult <> HTTP_ERROR then
    begin
        pszBuffer := PAnsiChar(GlobalLock(hgblBuffer));

        { Do something with the data and then unlock and
          release the handle when it is no longer needed }

        GlobalUnlock(hgblBuffer);
        GlobalFree(hgblBuffer);
    end;
  end;
```

It is important to remember to unlock and release the global memory handle when you are no longer using it. If you forget to release them, the application will have a memory leak.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or progress notifications during a file transfer. An event handler is simply a callback function which has a specific set of arguments, and the address of that function is passed to the ***RegisterEvent*** function in the library.

The first step to creating an event handler is to create a function which matches the signature of the callback function defined by the SocketTools event notification function. When the event handler is called, SocketTools will pass the handle to the client session, an event identifier to specify which event occurred, an error code value if an error occurred, and the user-defined value that was specified by the caller when the event was registered. In C++, the callback function would be declared as:

```
VOID EventHandler(HCLIENT hClient,
                  UINT nEventId,
                  DWORD dwError,
                  DWORD_PTR dwParam)
```

In Delphi, the equivalent function would be defined as:

```
procedure EventHandler(
        hClient: THandle;
        nEventId: Integer;
        dwError: LongWord;
        dwParam: LongWord); stdcall;
```

Note that the stdcall has been specified for the function. This is important, because the SocketTools libraries require the stdcall calling convention, as do any Windows API functions which use callback functions. Failure to specify the correct calling convention can prevent the callback function from being invoked or can cause the application to terminate abnormally.

Let's create an event handler that updates a progress bar as a file is being downloaded from an HTTP server. To do this, we'll create a procedure called **HttpEventHandler**:

```
procedure HttpEventHandler(
        hClient: THandle;
        nEventId: Integer;
        dwError: LongWord;
        dwParam: LongWord); stdcall;
var
   httpStatus: HTTPTRANSFERSTATUS;
   nPercent: Integer;
```

```
begin
    if nEventId = HTTP_EVENT_PROGRESS then
     begin
        HttpGetTransferStatus(hClient, httpStatus);
        nPercent := Trunc(100.0 * httpStatus.dwBytesCopied /
                                  httpStatus.dwBytesTotal);
        Form1.ProgressBar1.Position := nPercent;
    end;
end;
```

This event handler checks to see if the event ID indicates that it is a progress event, and if it is, calls **HttpGetTransferStatus** to determine the number of bytes that have been copied so far. This is used to calculate a percentage and a progress bar control is updated with that value.

Now that the event handler has been written, the next step is to register the event with the library so that it is called during the transfer. To do this, the **HttpRegisterEvent** function is called to enable notification for the progress event:

```
nResult := HttpRegisterEvent(
            hClient,
            HTTP_EVENT_PROGRESS,
            @HttpEventHandler,
            0);
```

The first argument to **HttpRegisterEvent** is the handle to the client session. The second argument is the event ID for which you want to enable notification, the third argument is the address of the callback function, and the fourth argument is a user-defined value. That same value is passed to the event handler as the dwParam argument.

Your event handler is now registered, and SocketTools will call your event handler during the process of downloading or uploading a file to notify you of the progress of the transfer.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PowerBASIC

The SocketTools Library Edition provides a complete set of function declarations and constants in the Include folder named **cstools12.inc** which can be included with your projects. The function declarations and examples included with SocketTools were developed using PowerBASIC 10.0 and PBForms 2.0. The function declarations are also compatible with the PowerBASIC 6.0 console compiler.

String Arguments

An important consideration when using the SocketTools libraries in PowerBASIC is how string arguments are used by the function. In most cases, the string is provided as input to the function, such as the hostname or address of a server to establish a connection with. However, in some cases the string is passed to the function as an output buffer into which the function copies data. An example of this would be the **HttpGetErrorString** function, which is used to obtain a description of a specific error. In C/C++ you would allocate a character array and pass it to the function, such as:

```
VOID ShowError()
{
  TCHAR szError[256];
  DWORD dwError;

  dwError = HttpGetLastError();
  if (dwError > 0)
  {
      HttpGetErrorString(dwError, szError, 256);
      MessageBox(NULL, szError, "Error", MB_OK);
  }
}
```

PowerBASIC has a string type called STRINGZ which is null-terminated, fixed-length string and is compatible with how strings are passed to the functions in the SocketTools API. Note that earlier versions of PowerBASIC called them ASCIIZ strings. The equivalent code would be:

```
SUB ShowError()
    LOCAL szError AS STRINGZ * 256
    LOCAL dwError AS DWORD

    dwError = HttpGetLastError()
    IF dwError <> 0 THEN
        HttpGetErrorString(dwError, szError, SIZEOF(szError))
        MSGBOX szError, %MB_OK OR %MB_ICONEXCLAMATION, "Error"
    END IF
END SUB
```

It is important to note that you should generally not use STRING types, since those represent variable length strings.

Byte Array Arguments

A number of SocketTools functions use byte arrays, either as an input argument to the function, or as an output argument which will contain data when the function returns. An example of this is the **HttpGetData** function, which will access a resource on the server and return the contents of that resource in a byte array passed to the function. For example, the following code in C++ would return the first 1024 bytes of the index page on a webserver:

```
BYTE byteBuffer[1024];
DWORD dwLength;
INT nResult;
```

```
dwLength = sizeof(byteBuffer);
nResult = HttpGetData(hClient,
                      "/index.html",
                      byteBuffer,
                      &dwLength,
                      0);
```

If the function is successful, the byteBuffer array will contain the first 1024 bytes of the index page. In PowerBASIC, the equivalent code would look like this:

```
DIM byteBuffer(1024) AS LOCAL BYTE
LOCAL dwLength AS DWORD
LOCAL nResult AS LONG

dwLength = SIZEOF(byteBuffer)
nResult = HttpGetData(hClient, _
                      "/index.html", _
                      BYREF byteBuffer(0), _
                      dwLength, _
                      0)
```

In this example, you will notice that the byteBuffer array is passed by reference, specifying the first element. The function will fill the byte array up to the number of bytes specified by the dwLength argument; when the function returns, dwLength will be updated with the actual number of bytes copied into the buffer.

If you wanted to convert the data in the byte array to a string type, you can terminate the data with a null character. However, this means that you should reserve a byte in the array. For example:

```
DIM byteBuffer(1024) AS LOCAL BYTE
LOCAL lpszBuffer AS STRINGZ PTR
LOCAL dwLength AS DWORD
LOCAL nResult AS LONG

dwLength = SIZEOF(byteBuffer) - 1 ' Reserve space for the null
nResult = HttpGetData(hClient, _
                      "/index.html", _
                      BYREF byteBuffer(0), _
                      dwLength, _
                      0)

IF nResult <> %HTTP_ERROR THEN
  lpszBuffer = VARPTR(byteBuffer(0))
  CONTROL SET TEXT hDlg, hCtl, @lpszBuffer
END IF
```

Note that you should only do this if the data returned by the function is actually text. In this example, it is acceptable to do because the byte array contains the HTML text for the index page.

Global Memory Handles

In addition to using byte arrays, some SocketTools functions can use global memory handles (HGLOBALs) to exchange large amounts of data. Using the Windows API, global memory handles are allocated by the **GlobalAlloc** function, dereferenced by the **GlobalLock** function and released by the **GlobalFree** function. These functions are defined in the WIN32API.INC module that is included with the language.

An application may choose to use a global memory handle instead of a pre-allocated buffer if the amount of data is very large, or the total amount of data that will be returned is unknown at the time the function is being called. Consider the call to the **HttpGetData** function used in the previous example. A pre-allocated buffer of 1024 bytes was passed to the function, and it copied up to that amount of data into the buffer. However, what if you wanted the complete page and did not know how large it was? You could attempt to determine the size

of the page that was being requested using the **HttpGetFileSize** function, and then use that value to allocate a buffer. However, this incurs additional overhead and it is not always possible to get the size of a resource on a web server. Another alternative would be to simply allocate a very large buffer, but this could result in the application allocating large amounts of memory that it doesn't use and you would still run the risk that it wouldn't be large enough.

The solution for this problem is to use a global memory handle rather than a pre-allocated buffer. Instead of copying the data into a buffer, the function allocates a global memory handle and stores the contents in the memory referenced by that handle. When the function returns, it passes the handle back to the caller. The caller then dereferences the handle to access the memory, and releases the handle when it is no longer needed. Here is an example of how it would be used in C/C++:

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
INT nResult;

nResult = HttpGetData(hClient,
                      "/index.html",
                      &hgblBuffer,
                      &dwLength,
                      HTTP_TRANSFER_CONVERT);

if (nResult != HTTP_ERROR)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Do something with the data and then unlock and
    // release the handle when it is no longer needed

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Note that the global memory handle is initialized to NULL and the length argument is initialized to zero. This is important to do because this is how the function knows that it should be returning a global memory handle instead of copying data into a buffer. If you forget to initialize those arguments, the function will fail and may cause the application to terminate with a general protection fault.

The equivalent code in PowerBASIC would look like this:

```
LOCAL hgblBuffer AS DWORD
LOCAL dwLength AS DWORD
LOCAL pszBuffer AS STRINGZ PTR
LOCAL nResult AS LONG

hgblBuffer = %NULL
dwLength = 0

nResult = HttpGetData(hClient, _
                      "/index.html", _
                      BYREF hgblBuffer, _
                      dwLength, _
                      %HTTP_TRANSFER_CONVERT)

IF nResult <> %HTTP_ERROR THEN
    pszBuffer = GlobalLock(hgblBuffer)

    ' Do something with the data and then unlock and
    ' release the handle when it is no longer needed
```

```
    GlobalUnlock(hgblBuffer)
    GlobalFree(hgblBuffer)
END IF
```

It is important to remember to unlock and release the global memory handle when you are no longer using it. If you forget to release them, the application will have a memory leak.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or progress notifications during a file transfer. An event handler is simply a callback function which has a specific set of arguments, and the address of that function is passed to the **HttpRegisterEvent** function in the library.

The first step to creating an event handler is to create a function which matches the signature of the callback function defined by the SocketTools event notification function. When the event handler is called, SocketTools will pass the handle to the client session, an event identifier to specify which event occurred, an error code value if an error occurred, and the user-defined value that was specified by the caller when the event was registered. In C++, the callback function would be declared as:

```
VOID CALLBACK EventHandler(
        HCLIENT hClient,
        UINT nEventId,
        DWORD dwError,
        DWORD_PTR dwParam)
```

In PowerBASIC, the equivalent function would be defined as:

```
SUB EventHandler STDCALL ( _
        BYVAL hClient AS LONG, _
        BYVAL nEventId AS LONG, _
        BYVAL dwError AS DWORD, _
        BYVAL dwParam AS DWORD)
```

Note that the STDCALL has been specified for the function. This is important, because the SocketTools libraries require this calling convention, as do any Windows API functions which use callback functions. Failure to specify the correct calling convention can prevent the callback function from being invoked or can cause the application to terminate abnormally.

Let's create an event handler that updates a progress bar as a file is being downloaded from an HTTP server. To do this, we'll create a subroutine called **HttpEventHandler**:

```
SUB HttpEventHandler STDCALL ( _
        BYVAL hClient AS LONG, _
        BYVAL nEventId AS LONG, _
        BYVAL dwError AS DWORD, _
        BYVAL dwParam AS DWORD)

    LOCAL httpStatus AS HTTPTRANSFERSTATUS
    LOCAL nPercent AS LONG
    LOCAL hDlg AS DWORD

    hDlg = dwParam ' Dialog with the progress bar control

    IF nEventId = %HTTP_EVENT_PROGRESS THEN
        HttpGetTransferStatus(hClient, httpStatus)
        nPercent = CLNG(100.0 * CDBL(httpStatus.dwBytesCopied) / _
                                CDBL(httpStatus.dwBytesTotal))
        CONTROL SEND hDlg, %IDC_PROGRESSBAR1, %PBM_SETPOS, nPercent, 0
    END IF
```

```
END SUB
```

This event handler checks to see if the event ID indicates that it is a progress event, and if it is, calls **HttpGetTransferStatus** to determine the number of bytes that have been copied so far. This is used to calculate a percentage and a progress bar control is updated with that value.

Now that the event handler has been written, the next step is to register the event with the library so that it is called during the transfer. To do this, the **HttpRegisterEvent** function is called to enable notification for the progress event:

```
nResult = HttpRegisterEvent( _
            hClient, _
            %HTTP_EVENT_PROGRESS, _
            CODEPTR(HttpEventHandler), _
            hDlg);
```

The first argument to **HttpRegisterEvent** is the handle to the client session. The second argument is the event ID for which you want to enable notification, the third argument is the address of the callback function, and the fourth argument is a user-defined value. That same value is passed to the event handler as the dwParam argument. To update a progress bar control or any other user interface object, the dwParam argument would typically be the handle to a dialog.

Your event handler is now registered, and SocketTools will call your event handler during the process of downloading or uploading a file to notify you of the progress of the transfer.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Library Overview

The SocketTools Library Edition includes libraries that implement fourteen standard Internet application protocols, as well as libraries which provide support for general TCP/IP networking services, encoding and compressing files, processing email messages and ANSI terminal emulation. The following libraries are included in the SocketTools Library Edition:

Domain Name Service

The Domain Name Service (DNS) protocol is what applications use to resolve domain names into Internet addresses, as well as provide other information about a domain, such as the name of the mail servers which are responsible for receiving email for users in that domain. The DNS library enables an application to query one or more nameservers directly, without depending on the configuration of the client system.

Encoding, Compression and Encryption

The Encoding, Compression and Encryption library provides functions for encoding and decoding binary files, typically attachments to email messages. The process of encoding converts the contents of a binary file to printable text. Decoding reverses the process, converting a previously encoded text file back into a binary file. The library supports a number of different encoding methods, including support for the base64, uucode, quoted-printable and yEnc algorithms. The library can also be used to encrypt data using AES-256 bit encryption, and compress data in a user-supplied buffer or in a file.

File Transfer Protocol

The File Transfer Protocol (FTP) library provides functions for uploading and downloading files from a server, as well as a variety of remote file management functions. In addition to file transfers, an application can create, rename and delete files and directories, list files and search for files using wildcards. The library provides both high level functions, such as the ability to transfer multiple files in a single function call, as well as access to lower level remote file I/O functions. This API supports secure file transfers using FTPS (FTP+TLS) and SFTP (FTP+SSH).

GeoIP Location

The Web Services library provides an API for obtaining geographical information about the physical location of the computer system based on its external IP address. This can enable developers to know where their application is being used, and provide convenience functionality such as automatically completing a form based on the location of the user.

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) library provides an interface for accessing documents and other types of files on a server. In some ways it is similar to the File Transfer Protocol in that it can be used to upload and download files; however, the protocol has expanded to also support remote file management, script execution and distributed authoring over the World Wide Web. The SocketTools Hypertext Transfer Protocol library implements version 0.9, 1.0 and 1.1 of the protocol, including features such as support for proxy servers, persistent connections, user-defined header fields and chunked data.

Internet Control Message Protocol

The Internet Control Message Protocol (ICMP) is commonly used to determine if a server is reachable and how packets of data are routed to that system. Users are most familiar with this protocol as it is implemented in the ping and tracert command line utilities. The ping command is used to check if a system is reachable and the amount of time that it takes for a packet of data to make a round trip from the local system, to the server and then back again. The tracert command is used to trace the route that a packet of data takes from the local system to the server, and can be used to identify potential problems with overall throughput and latency. The library can be used to build in this type of

functionality in your own applications, giving you the ability to send and receive ICMP echo datagrams in order to perform your own analysis.

Internet Message Access Protocol

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP. The SocketTools IMAP library implements the current standard for this protocol, and provides functions to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The API is designed as a superset of the Post Office Protocol API, so developers who are used to working with the POP3 library will find the IMAP library very easy to integrate into an existing application.

Large Language Model Client

The Large Language Model (LLM) client library enables your application to interact with modern AI services to generate text, answer questions, and process natural language input. The library provides a common interface that can be used with both cloud-based services and locally hosted models, without requiring your application to work directly with each provider's API. The library handles the provider-specific details of creating requests, managing connections, formatting prompts, and returning responses. This allows your application to focus on how the generated content is used, rather than the details of how each provider implements its service.

Mail Messages

The Mail Message (MIME) library provides an interface for composing and processing email messages and newsgroup articles which are structured according to the Multipurpose Internet Mail Extensions (MIME) standard. Using this library, an application can easily create complex messages which include multiple alternative content types, such as plain text and styled HTML text, file attachments and customized headers. It is not required that the developer understand the complex MIME standard; a single function call can be used to create multipart message, complete with a styled HTML text body and support for international character sets. The Mail Message library can be easily integrated with the other mail related protocol libraries, making it extremely easy to create and process MIME formatted messages.

Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) is used with servers that provide news services. This is similar in functionality to bulletin boards or message boards, where topics are organized hierarchically into groups, called newsgroups. Users can browse and search for messages, called news articles, which have been posted by other users. On many servers, they can also post their own articles which can be read by others. The largest collection of public newsgroups available is called USENET, a world-wide distributed discussion system. In addition, there are a large number of smaller news servers. For example, Microsoft operates a news server which functions as a forum for technical questions and announcements. The SocketTools library provides a comprehensive interface for accessing newsgroups, retrieving articles and posting new articles. In combination with the Mail Message library to process the news articles, SocketTools can be used to integrate newsgroup access with an existing email application, or you can implement your own full-featured newsgroup client.

News Feeds

The News Feed (RSS) library enables an application to download and process a syndicated news feed in standard format. News feeds can be accessed remotely from a web server, or locally as an XML formatted text file. The source of the feed is determined by the URI scheme that is specified. If the http

or https scheme is specified, then the feed is retrieved from a web server. If the file scheme is used, the feed is considered to be local and is accessed from the disk or local network. The News Feed library provides an API that enables you to open a feed by URL and iterate through each of the items in the feed or search for a specific feed item. The library also provides a function that can be used to parse a string that contains XML data in RSS format, where the feed may have been retrieved from other sources such as a database.

Post Office Protocol

The Post Office Protocol (POP3) provides access to a user's new email messages on a mail server. Functions are provided for listing available messages and then retrieving those messages, storing them either in files or in memory. Once a user's messages have been downloaded to the local system, they are typically removed from the server. This is the most popular email protocol used by Internet Service Providers (ISPs) and the SocketTools library provides a complete interface for managing a user's mailbox. This library is typically used in conjunction with the Mail Message library, which is used to process the messages that are retrieved from the server.

Remote Commands

The Remote Command library is used to execute a command on a server and return the output of that command to the client. The SocketTools library provides an interface to this protocol, enabling applications to remotely execute a command and process the output. This is most commonly used with UNIX based servers, although there are implementations of remote command servers for the Windows operating system. The SocketTools library supports both the rcmd and rshell remote execution protocols and provides functions which can be used to search the data stream for specific sequences of characters. This makes it extremely easy to write Windows applications which serve as light-weight client interfaces to commands being executed on a UNIX server or another Windows system. The library can also be used to establish a remote terminal session using the rlogin protocol, which is similar to how the Telnet protocol functions.

Secure Shell Protocol

The Secure Shell (SSH) protocol is used to establish a secure connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The SSH library provides an API for establishing the connection and handling the standard I/O functions needed by the program. The library also provides functions that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server.

Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The library provides an API for addressing and delivering messages, and extended features such as user authentication and delivery status notification. Unlike Microsoft's Messaging API (MAPI) or Collaboration Data Objects (CDO), there is no requirement to have certain third-party email applications installed or specific types of servers installed on the local system. The SocketTools library can be used to deliver mail through a wide variety of systems, from standard UNIX based mail servers to Windows systems running Exchange or Lotus Notes and Domino. Using the SocketTools library, messages can be delivered directly to the recipient, or they can be routed through a relay server, such as an Internet Service Provider's mail system. The Mail Message library can be integrated with this library in order to provide an extremely simple, yet flexible interface for composing and delivering mail messages.

SocketWrench

The SocketWrench library provides a higher-level interface to the Windows Sockets API, designed to be suitable for programming languages other than C and C++. If needed, function calls can be

intermixed between the SocketWrench and Windows Sockets libraries. In addition, the SocketWrench supports secure communications using Transport Layer Security (TLS).

Telnet Protocol

The Telnet protocol is used to establish a connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The Telnet library provides an API for establishing the connection, negotiating certain options (such as whether characters will be echoed back to the client) and handling the standard I/O functions needed by the program. The library also provides functions that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. This library can be combined with the Terminal Emulation library to provide complete terminal emulation services for a standard ANSI or DEC-VT220 terminal.

Terminal Emulation

The Terminal Emulation library provides a comprehensive API for emulating an ANSI or DEC-VT220 character terminal, with full support for all standard escape and control sequences, color mapping and other advanced features. The library functions provide both a high level interface for parsing escape sequences and updating a display, as well as lower level primitives for directly managing the virtual display, such as controlling the individual display cells, moving the cursor position and specifying display attributes. This library can be used in conjunction with the Remote Command or Telnet Protocol library to provide terminal emulation services for an application, or it can be used independently. For example, this library could also be used to provide emulation services for a program that provides serial modem connections to a server.

Text Messaging

The Text Message library enables applications to send text messages to mobile devices. It provides an interface that can be used to obtain information about the wireless service provider that is associated with the phone number for a smartphone or other mobile device, and can send a message with a single function call. Messages can be delivered directly to the service provider's gateway, or can be relayed through a local mail server. With this API, an application can send text message alerts when certain conditions occur (such as an error) or as a notification mechanism that's used in addition standard email messages.

Network Time Protocol

The Network Time Protocol library provides an interface for synchronizing the local system's time and date with that of a server. The library enables developers to query a server for the current time and then update the system clock if desired.

Web Storage

The Web Storage library provides a private cloud storage API for uploading and downloading shared data files which are available to your application. This is primarily intended for use by developers to store configuration information and other data generated by the application. For example, you may want to store certain application settings, and the next time a user or organization installs your software, those settings can be downloaded and restored.

WebView Browser

The WebView browser library provides a standard Win32-style API for creating and controlling an instance of the Microsoft Edge WebView2 control. It gives developers a simpler procedural API for embedding modern web content without requiring them to work directly with COM interfaces, event handlers and WebView2 initialization details. Applications can use the API to attach a WebView2 instance to an existing window, create a standalone browser window or manage navigation, scripting, messaging, document saving and other browser features through familiar function calls. This makes it

suitable for traditional GUI applications as well as console applications that need to create and control a browser instance from a command-line interface.

Whois Protocol

The Whois protocol library provides an interface for requesting information about an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The Whois protocol enables an application to query a server which provides that registration information. The SocketTools library provides an API for requesting that information and returning it to the program so that it can be displayed or processed.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service

The Domain Name Service (DNS) protocol is what applications use to resolve domain names into Internet addresses, as well as provide other information about a domain, such as the name of the mail servers which are responsible for receiving email for users in that domain. All of the SocketTools libraries provide basic domain name resolution functionality, but the Domain Name Services library gives an application direct control over what servers are queried, the amount of time spent waiting for a response and the type of information that is returned.

The first step that your application must take is to initialize the library and then create a handle for the client session. Unlike many of the other libraries, there are no connection related functions because DNS uses UDP datagrams rather than TCP streams. The following functions are available for use by your application:

DnsInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other DNS API functions.

DnsCreateHandle
Create a handle that is used by the library to reference the client session. There are two arguments, a timeout period and a retry count. These values are used together to determine the total amount of time that the library will take in an attempt to resolve a hostname or IP address. The default values of 15 seconds and 4 retries are recommended for most applications.

DnsCloseHandle
Release the handle that was previously created by the call to **DnsCreateHandle**. Any memory allocated by the library on behalf of the application is released and the datagram socket that was created is closed.

DnsRegisterServer
Specify a nameserver that the library should use to resolve queries. By default, the library will use the nameservers that the local host was configured to use. This function enables you to override that default and specify your own servers, independent of the system's configuration.

DnsUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Address Conversion

Internet Protocol (IP) addresses can be represented in one of two ways, either as unsigned 32-bit integer value or as string where each byte of the address is written as an integer value and separated by periods. For example, the local loopback IP address can either be specified as the string "127.0.0.1" or as the integer value 16777343. In most cases, using the string form of the address is easier; however, some functions require that the numeric value be used. The following functions are provided to enable you to convert between the two formats.

DnsGetAddress
Convert an IP address string in dotted notation into a 32-bit integer value.

DnsFormatAddress
Convert a numeric IP address into a string in dotted notation, copying the result into a buffer that you provide to the function.

Host Tables

When resolving a host name or IP address, the library will first search the local system's host table, a file that is used to map host names to addresses. On Windows 95/98 and Windows Me, if the file exists it is usually found

in C:\Windows\hosts. On Windows NT and later versions, it is found in C:\Windows\system32\drivers\etc\hosts. Note that the file does not have an extension.

DnsGetDefaultHostFile
Return the full path of the file that contains the default host table for the local system. This can be useful if you wish to temporarily switch between the default host file and another host file specific to your application.

DnsGetHostFile
Return the full path of the host table that is currently being used by the library. Initially this is the same as the default host table for the local system.

DnsSetHostFile
Specify a new host table which the library should use to resolve host names and IP addresses. This can be used by an application to provide its own local cache of host names and addresses in order to speed up the process of host name resolution.

Host Name Resolution

The library can be used to resolve host names into IP addresses, as well as perform reverse DNS lookups converting IP addresses into the host names that are assigned to them. The library will search the local system's host table first, and then perform a nameserver query if required.

DnsGetHostAddress
Resolve a host name into an IP address, returned as a string in dotted notation. The library first checks the system's local host table, and if the name is not found there, it will perform a nameserver query for the A (address) record for that host.

DnsGetHostName
Resolve an IP address into a host name. The address is passed as a string in dotted notation, and the fully qualified host name is returned in a string buffer you provide to the function. The library first checks the system's local host table, and if the address is not found there, it will perform a nameserver query for the PTR (pointer) record for that address.

Mail Exchange Records

When a system needs to deliver a mail message to someone, it needs to determine what server is responsible for accepting mail for that user. This is done by looking up the mail exchange (MX) record for the domain. For example, if a message was addressed to joe@example.com, to determine the name of the mail server that would accept mail for that recipient, you would perform an MX record query against the domain example.com. A domain may have more than one mail server, in which case multiple MX records will be returned.

DnsEnumMailExchanges
Enumerate all of the mail exchanges for the specified domain. If there are multiple servers, they may be prioritized so that certain servers are given precedence over the others. This function will return the highest priority servers first in the list. Refer to the Technical Reference for an example of how this function can be used.

Advanced Queries

In addition to providing host name and IP address resolution, the library can be used to perform advanced queries for other types of records.

DnsGetHostInfo
Return additional information about the specified host name. If the name server has been configured to provide host information for the domain, this function will return that data. Typically it is used to indicate what hardware and operating system the host uses.

DnsGetHostServices

Return information about the UDP and TCP based services that the host provides. If defined, this will return a list of service names such as "ftp" and "http". Note that your application should not depend on this information to be a definitive list of what services a server provides.

DnsGetRecord

Perform a general nameserver query for a specific record type. This function can be used to perform queries for the common record types such as A and PTR records, as well as for other record types such as TXT (text) records. Refer to the Technical Reference for more information about the specific types of records that can be returned.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Encoding, Compression and Encryption

A common requirement for applications which use Internet protocols is the need to encode binary files, as well as compress data to reduce the bandwidth and time required to send or receive the data. Encoding a binary file converts the contents of the file into printable characters which can be safely transferred over the Internet using protocols that only support a subset of 7-bit ASCII characters. This is commonly a restriction for email, since many mail servers still are not capable of correctly processing messages which contain control characters, 8-bit data or multi-byte character sequences found in International text.

To address this problem, the sender encodes and sends the data as part of a message; the recipient then extracts and decodes the data, with the end result being the same as the original, without any potential corruption by the mail servers which store and/or forward the message. The File Encoding library supports several encoding and decoding methods, including standard base64 encoding, quoted-printable encoding and uuencoding. For applications which access USENET newsgroup, the library also supports the newer yEnc encoding method which has become a popular method for attaching binary files to a message.

This library also can be used to compress files, reducing their overall size. Two compression algorithms are supported, the standard deflate algorithm which is commonly used in Zip files, and an algorithm based on the Burrows-Wheeler Transform (BWT) which can offer improved compression over the deflate algorithm for some types of files. The developer has control over the type of compression performed, as well as details such as the level of compression which determines how much memory and CPU time is allocated to compress the data. Developers can even create their own custom compression formats by creating an application-specific header block, typically represented by a structure or user-defined type that can be used to provide information to the program.

Unlike the other SocketTools libraries, there are no initialization functions for this library, and there are no handles used. All operations are performed either on files or on memory buffers provided by the application. The library is split into two general areas of functionality. The first group of functions enables you to encode and decode binary files and the second group enables you to compress and expand data.

Note that if you are interested in using this library for purposes of attaching files to an email message, it is not necessary that you use these functions. The Mail Message library has the ability to automatically encode and decode file attachments without requiring that you use the functions in this library. However, the File Encoding library is useful if you need the ability to encode, compress and/or encrypt data for use in other applications.

Encoding Types

There are several different encoding types available, with the default being the standard MIME encoding called Base64. The following encoding methods are supported by the library:

Base64

Base64 encoding works by representing three bytes of data as four printable characters. Each of the three bytes is converted into four six-bit numbers, and each six-bit number is converted to one of 64 printable characters (which is where the encoding method gets its name). Base64 is the default encoding method used by the library and is the standard encoding used for MIME formatted email messages as well as many other applications.

Quoted-Printable

Quoted-printable encoding is primarily used in email messages, and is best used when the data being encoded is text which consists primarily of printable characters. Only characters with the high-bit set or a certain subset of printable characters are actually encoded by representing them as their hexadecimal value. All other printable characters are passed through unmodified.

Uucode

One of the original encoding methods used for email, it gets its name from two UNIX command-line utilities called uuencode and uudecode, which were used to encode and decode files. Like Base64, uuencoding converts three bytes of data into four six-bit numbers, and then a value of 32 is added to ensure that it is printable. Uuencoding also adds some additional characters which are used to ensure the integrity of the encoded data. This encoding method is still used when posting files to USENET newsgroups, but has largely been replaced by Base64 when attaching files to email messages.

yEnc

yEnc is an encoding method that was created specifically for binary newsgroups on USENET. Because USENET doesn't have the same limitations as email systems in terms of what kind of characters can be safely used, yEnc only encodes null characters and certain control characters; the remaining 8-bit data is passed through as is which can significantly reduce the overall size of the encoded data. yEnc also uses checksums to ensure the integrity of the data and is designed so that a large file can be split across multiple messages and then recreated.

Data Encoding

Encoding a binary file converts the contents of the file into printable characters which can be safely transferred over the Internet using protocols that only support a subset of 7-bit ASCII characters. This is commonly a restriction for email, since many mail servers still are not capable of correctly processing messages which contain control characters, 8-bit data or multi-byte character sequences found in International text. To address this problem, the sender encodes and sends the data as part of a message; the recipient then extracts and decodes the data, with the end result being the same as the original, without any potential corruption by the mail servers which store and/or forward the message.

EncodeFile

This function encodes a file using the specified encoding method, storing the encoded data in a new file. An option also allows you to automatically compress the data prior to encoding it in order to reduce the overall size of the encoded file.

DecodeFile

This function decodes a previously encoded file using the specified encoding method, restoring the original contents. If the encoded data was compressed, this function can also be used to automatically expand the data after it has been decoded.

EncodeBuffer

This function encodes a block of data in memory using the specified encoding type. This is similar to the **EncodeFile** function, except that instead of using disk files, all of the encoding is done in memory. As with encoding a file, you can also specify that you want the data to be compressed prior to being encoded.

DecodeBuffer

This function decodes a previously encoded block of data. This is similar to the **DecodeFile** function, except that instead of using disk files, all of the decoding is done in memory. As with decoding a file, you can also specify that you want compressed data to be automatically expanded after it has been decoded.

IsUnicodeText

This function can be used to check the contents of a string to ensure that it contains valid UTF-8 or UTF-16 encoded Unicode text. One of the primary uses of the function is to determine if data received over the network uses valid UTF-8 character encoding. The **UnicodeDecodeText** function can then be used to convert it to UTF-16 or multi-byte text.

UnicodeDecodeText

This function decodes UTF-8 encoded text and returns the decoded text in a string provided by the

caller. It can be used to convert UTF-8 text to UTF-16, or to localized multi-byte text. It is similar to the **MultiByteToWideChar** function, but performs some additional checks to ensure the encoded text is valid and can be safely decoded.

UnicodeEncodeText
This function encodes a multi-byte or Unicode string as UTF-8 encoded text. It is similar to the **WideCharToMultiByte** function and ensures Unicode text is normalized prior to being encoded.

Data Compression

In addition to encoding and decoding data, the library can be used to compress data in order to reduce its size. The compression functions may be used separately, or may be used as part of the process of encoding a file or a block of data.

CompressFile
This function reduces the size of a file using the standard Deflate algorithm. This is the same algorithm that is commonly used in Zip archives. Note however, that this does not create a Zip file, it simply uses the same compression method.

ExpandFile
This function restores the original contents of a file that was previously compressed using the **CompressFile** function. Note that this function is not designed to extract files from a Zip archive or expand data compressed using a different algorithm.

CompressBuffer
This function uses the Deflate algorithm to reduce the size of a block of data. This is similar to the **CompressFile** function except that it performs the compression on data in memory rather than in a disk file.

ExpandBuffer
This function restores the data that was previously compressed using the **CompressBuffer** function.

There are some additional functions for compressing files that provide more advanced options such as the ability to specify the compression type and level, as well as enabling you to create your own custom file compression formats. Please refer to the Technical Reference for more information.

Data Encryption

There are several functions which provide a simplified interface for encrypting and decrypting data using the AES-256 Advanced Encryption Standard algorithm.

AesEncryptFile
This function encrypts the contents of a file. A password string can be provided which is used to generate the encryption key. To encrypt data stored in memory, use the **AesEncryptBuffer** function.

AesDecryptFile
This function is used to decrypt the contents of a file that was previously encrypted using **AesEncryptFile**.

AesEncryptBuffer
This function is used to encrypt the contents of a buffer in memory. The encrypted data is returned in a buffer that is provided by the caller. The encrypted data will be in a binary format which can contain embedded null bytes. If you need to encrypt string values, it's recommended you use the **AesEncryptString** function.

AesDecryptBuffer
This function is used to decrypt a buffer in memory that was previously encrypted using **AesEncryptBuffer**.

AesEncryptString

This function is used to encrypt a null-terminated string and the encrypted data is returned as base64 encoded text. This is similar to the **AesEncryptBuffer** function, but it is designed to work specifically with strings and the encrypted output can be safely stored as a text value by the application.

AesDecryptString

This function is used to decrypt a string that was previously encrypted using **AesEncryptString**.

The encryption functions use the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. It is important to note the availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol

The File Transfer Protocol (FTP) is the most common application protocol used to upload and download files between a local system and a server. In addition to basic file transfer capabilities, FTP also enables a client application to perform common file and directory management functions on the server, such as renaming and deleting files or creating new directories. The SocketTools Library Edition also supports secure file transfers using SSH (SFTP) and TLS (FTPS) by simply specifying an option when establishing the connection.

The first step that your application must take is to initialize the library and then establish a connection. The following functions are available for use by your application:

FtpInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other FTP API functions.

FtpConnect
Connect to the server, using either a host name or IP address. The function has several options related to security as well as the general operation of the library. One important option is FTP_OPTION_PASSIVE, which instructs the library to use passive mode file transfers. If the local system is behind a firewall or a route which uses Network Address Translation (NAT), it is often necessary to use this option. This function returns a client handle which is used in subsequent calls to the library.

FtpProxyConnect
A variation on the standard connection, this function can be used to connect to an FTP server through a proxy server. The library provides support for a number of standard proxy types, such as those used by the Gauntlet and InterLock proxy servers. A custom proxy server type is also supported where your application can send any custom commands required to establish the connection.

FtpLogin
Authenticate the client session, providing the server with a user name, password and optionally an account name. It is also possible to use an anonymous (unauthenticated) session by providing empty strings as the username and password.

FtpDisconnect
Disconnect from the server and release the memory allocated for that client session. After this function is called, the client handle is no longer valid.

FtpUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

File Transfers

The library provides several functions which can be used to transfer files between the local and server. This group of functions is high level, meaning that it is not necessary to actually write the code to read and/or write the file data. The library automatically handles the lower level file I/O and notifies your application of the status of the transfer by periodically generating progress events.

FtpGetData
This function transfers a file from the server to the local system, storing the file data in memory. This can be useful if your application needs to perform some operation based on the contents of the file, but does not need to store the file locally.

FtpGetFile
This function transfers a file from the server and stores it in a file on the local system. This function is similar to how the GET command works for the command-line FTP client in Windows.

FtpGetMultipleFiles
This function transfers multiple files from the server and stores them in a directory on the local system. A wildcard may be specified so that only files which a certain name or those that match a particular file extension are downloaded. This function is similar to how the MGET command works for the command-line FTP client in Windows.

FtpPutData
This function creates a file on the server containing the data that you provide. This can be useful if your application wants to upload dynamically created content without having to create a temporary file on the local system.

FtpPutFile
This function uploads a file from the local system to the server. This function is similar to how the PUT command works for the command-line FTP client in Windows.

FtpPutMultipleFiles
This function transfers multiple files from the local system to a directory on the server. A wildcard may be specified so that only files with a certain name or those that match a particular file extension are uploaded. This function is similar to how the MPUT command works for the command-line FTP client in Windows.

File Management

In addition to performing file transfers, the File Transfer Protocol library can also perform many of the same kinds of file management functions on the server as you would on the local system.

FtpDeleteFile
Delete a file from the server. This operation requires that the current user have the appropriate permissions to delete the file.

FtpRenameFile
Change the name of a file or move a file to a different directory. This operation requires that the current user have the appropriate permissions to rename the file. If the file is being moved to another directory, the user must have permission to access that directory.

FtpGetFileStatus
Return status information about the file in the form of a structure. This typically specifies the ownership, access permissions, size and modification time for the file. It is similar to opening a directory on the server and reading information about the file, but with less overhead.

FtpGetFileSize
Return the size of a file on the server without actually downloading the contents of the file.

FtpGetFileTime
Return the modification time for the specified file on the server. This can be used by you application to determine if the file has been changed since the time that you last uploaded or downloaded the contents.

FtpSetFileTime
Update the modification time for a file on the server. This function requires that the current user have the appropriate permissions to change the last modification timestamp for the file. Note that this is not supported on all servers and in some cases may be restricted to specific accounts.

FtpGetFilePermissions
Return the access permissions for a file on the server. This can be used to determine if a file can be read, modified and/or deleted by the current user. For users who are familiar with UNIX file permissions, it is the same type which is used by the library.

FtpSetFilePermissions

Change the access permissions for a file. This function is supported on most UNIX based servers, as well as any other server that supports the site-specific CHMOD command.

Directory Management

The library also provides a set of functions which can be used to access and manage directories or folders, including the ability to list and search for files, create new directories and remove empty directories from the server.

FtpOpenDirectory

Open the specified directory on the server. This is the first step in returning a list of files in the directory. After the directory has been opened, information about the files it contains can be returned to the application. The directory path may also include wildcards to only return information about a certain subset of files based on the file name or extension.

FtpGetFirstFile

Return information about the first file in the directory that has been opened. This is similar to how the Windows API function **FindFirstFile** works.

FtpGetNextFile

Return information about the next file in the directory that has been opened. This function is called repeatedly until it indicates that all of the files have been returned. This is similar to how the Windows API function **FindNextFile** works.

FtpChangeDirectory

Change the current working directory on the server. This is similar to how the CD command is used from the command-line to change the current directory in Windows. If a path is not specified in the file name, the current working directory is where files will be uploaded to and downloaded from.

FtpCreateDirectory

Create a new directory on the server. This requires that the current user have the appropriate access permissions in order to create the directory.

FtpRemoveDirectory

Remove an empty directory from the server. This operation requires that the current user have the appropriate permissions to delete the directory. For safety, it is required that the directory does not contain any files or subdirectories or the operation will fail.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GeoIP Location Service

The Web Services library provides an API for obtaining geographical information about the physical location of the computer system based on its external IP address. This can enable developers to know where their application is being used, and provide convenience functionality such as automatically completing a form based on the location of the user.

The connection to the location service is always secure and does not require you subscribe to any third-party services. The accuracy of this information can vary depending on the location, with the most detailed information being available for North America. The country and time zone information for all locations is generally accurate. However, as the location information becomes more precise, details such as city names, postal codes and specific geographic locations (e.g.: longitude and latitude) may have reduced accuracy.

Software which is designed to protect the privacy of users, such as those which route all Internet traffic through proxy servers or VPNs, can significantly impact the accuracy of this information. In this case, the data returned in this structure may reflect the location of the network or proxy server, and not the location of the person using your application. It is recommended you always request permission from the user before acquiring their location, have them confirm the location is correct and provide a mechanism for them to update the information.

Functions

To obtain the location of the local computer system, use the following functions:

WebInitialize
Initialize the library for the current process. This must be the first function call the application makes before calling the other service API functions.

WebGetLocation
This function populates a WEB_LOCATION structure which contains information about the physical location of the system. This structure has a number of members which provide details about the location:

Member	Description
szLocationId	A string which contains contains a string of hexadecimal characters which uniquely identifies the location for this computer system. This value is used internally by the location service, and may also be used by the application for its own purposes.
szIPAddress	A string which contains the external IP address for the local system. If the system has been assigned multiple IP addresses, it reflects the address of the interface used to establish the connection with the location server.
nAutonomousSytemNumber	An integer which is used to uniquely identify a global network (autonomous system) which is connected to the Internet. This value can be used to determine the ownership of a particular network.
szOrganization	A string which identifies the organization associated with the local system's external IP address. For residential end-users this is typically the name of their Internet Service provider, however it may also identify a private company.
szRegionName	A string which identifies a broad geographical area, such

	as "North America" or "Southeast Asia".
nRegionCode	An integer which identifies the geographical region. This value corresponds to standard UN M49 region codes.
szCountryName	A string which contains the full name of the country in which the external IP address is located, such as "United States". These names will always be in English, regardless of the current system locale.
szCountryAlpha2	A string which contains the ISO 3166-1 alpha-2 code assigned to the country. For example, the alpha-2 code for the United States is "US".
szCountryAlpha3	A string which contains the ISO 3166-1 alpha-3 code assigned to the country. For example, the alpha-3 code for the United States is "USA".
nCountryCode	An integer value which identifies the country using the standard UN country code. For example, the numeric country code for the United States is 840.
szSubdivision	A string which identifies a geopolitical subdivision within a country. In the United States, this will contain the full name of the state or commonwealth. In Canada, this will contain the name of the province or territory.
szSubdivisionCode	A string which is either a two- or three-letter code which identifies a geopolitical subdivision within the country. These codes are defined by the ISO 3166-2 standard. For example, the code for the state of California in the United States is "CA".
szCityName	A string which identifies the city at this location. These names will always be in English, regardless of the current system locale. If the city name cannot be determined, this member may contain an empty string.
szPostalCode	A string which contains the postal code associated with the location. In the United States, this is a 5-digit numeric code. Local delivery portions of a postal code (such as the ZIP+4 code in the United States) are not included.
szCoordinates	A string which specifies the location expressed using the Universal Transverse Mercator (UTM) coordinate system with the WGS-84 ellipsoid. These coordinates are commonly used with the Global Positioning System (GPS).
szTimezone	A string which specifies the full time zone name. These names are defined by the Internet Assigned Numbers Authority (IANA) and have values like "America/Los_Angeles" and "Europe/London".
szTzShortName	A string which specifies the abbreviated time zone code. If daylight savings time is used within the time zone, then this value can change based on whether or not daylight savings is in effect. If a short time zone code cannot be

	determined, a value such as "UTC+9" may be returned, indicating the number of hours ahead or behind UTC.
nTimezoneOffset	An integer which specifies the number of seconds east or west of the prime meridian (UTC). A positive value indicates a time zone which is east of the prime meridian and a negative value indicates a time zone which is west of the prime meridian.
dLatitude	A real number which specifies the latitude of the location in decimal format. A positive value indicates a location which is north of the equator, while a negative value is a location which is south of the equator.
dLongitude	A real number which specifies the longitude of the location in decimal format. A positive value indicates a location which is east of the prime meridian, while a negative value is a location which is west of the prime meridian.
stLocalTime	A SYSTEMTIME structure which contains information about the current date and time at the location, adjusted for its time zone and whether or not it's in daylight savings time.

WebUninitialize

Release all resources that were allocated for the current process. This is the last function call the application should make prior to terminating.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is the most prevalent application protocol used on the Internet today. It was originally used for document retrieval, and has grown into a complex protocol which supports file uploading, script execution, file management and distributed web authoring through extensions such as WebDAV. The SocketTools Hypertext Transfer Protocol library implements version 0.9, 1.0 and 1.1 of the protocol, including features such as support for proxy servers, persistent connections, user-defined header fields and chunked data.

File Transfers

Similar to the API used with the File Transfer Protocol library, you can use HTTP to upload and download files. In addition to the standard method for downloading files, the library supports two methods for uploading files, using either the PUT or the POST command. When downloading a file from the server, you can either store the contents in a local file, or you can copy the data into a memory buffer that you allocate. Similarly, when uploading files, you can either specify a local file to upload, or you can provide a memory buffer that contains the file data to send to the server. High level functions such as **HttpPutFile** and **HttpGetFile** can be used to transfer files in a single function call. There are also functions such as **HttpOpenFile** and **HttpCreateFile** which provide lower level file I/O interfaces.

Script Execution

Another common use for HTTP is to execute scripts on the web server. The application can pass additional data to the script, which is similar in concept to how arguments are passed to a command that is entered from the command prompt. This typically uses the standard GET and POST commands, and the resulting output from the script is returned back to the application where it can be displayed or processed. There are several different functions you can use, depending on the service you're requesting information from. The **HttpPostData** function is general-purpose and can be used with virtually any type of POST request, including submitting binary data. However, it can require some additional settings such as the content type and encoding method. The **HttpPostJson** and **HttpPostXml** functions are specifically designed for requests which require JSON or XML payloads, respectively.

Uniform Resource Locators

Anyone who has used a web browser is familiar with the Uniform Resource Locator (URL); it is the value that is entered as the address of a website. URLs have a specific format which provides information about the server, the port number and the name of the resource that is being accessed:

[http|https]://[***username*** : [***password***] @] hostname [:***port***] / resource [? ***parameters***]

The first part of the URL identifies the protocol, also known as the scheme, which will be used. With web servers, this will be either http or https for secure connections. If a username and password is required for authentication, this can be included in the URL before the name of the server. Next, there is the name of the server to connect to, optionally followed by a port number. If no port number is given, then the default port for the protocol will be used. This is followed by the resource, which is usually a path to a file or script on the server. Parameters to the resource may also be specified, called the query string, which are typically used as arguments to a script that is executed on the server.

Understanding how a URL is constructed will help in understanding how the different functions in the library work together. For example, the server name and port number portion of the URL are the values passed to the **HttpConnect** function to establish the connection. The user name and password values are passed to the **HttpAuthenticate** function to authenticate the client session. And the resource name is passed to the **HttpGetData** or **HttpGetFile** functions to transfer it to the local system.

The first step that your application must take is to initialize the library and then establish a connection. The following functions are available for use by your application:

HttpInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other HTTP API functions.

HttpConnect
Establish a connection to the server. This function will return a handle to a client session which is used in subsequent calls to the HTTP API.

HttpProxyConnect
A variation on the standard connection, this function can be used to connect to an HTTP server through a proxy server. The library provides support for standard CERN type proxies as well as tunneling proxies that are commonly used with secure TLS connections.

HttpAuthenticate
If the server requires the client to authenticate prior to accessing a resource, this function can be called to provide the user name and password. This is commonly used to restrict access to certain areas of a website to authenticated users only. Although it is permitted to authenticate immediately after connecting to a server, it is not required. An application can wait until the server returns an error indicating that authentication is required to access the resource. It can call this function at that time, and then re-request the resource. This is how most browsers work. This function may be called more than once during a session if the client needs to change the current user name and/or password being used to authenticate access to the server.

HttpDisconnect
Disconnect from the server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

HttpUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

File Transfers

Using an API similar to the File Transfer Protocol library, this library provides several functions which can be used to transfer files between the local and server. This group of functions is high level, meaning that it is not necessary to actually write the code to read and/or write the file data. The library automatically handles the lower level file I/O and notifies your application of the status of the transfer by periodically generating progress events.

HttpGetData
This function transfers a file from the server to the local system, storing the file data in memory. This can be useful if your application needs to perform some operation based on the contents of the file, but does not need to store the file locally.

HttpGetFile
This function transfers a file from the server and stores it in a file on the local system.

HttpPutData
This function creates a file on the server containing the data that you provide. This can be useful if your application wants to upload dynamically created content without having to create a temporary file on the local system.

HttpPutFile
This function uploads a file from the local system to the server using the PUT command. Not all

servers support this command, and some may require that the client authenticate prior to calling this function.

HttpPostFile
This function uploads a file from the local system to the server using the POST command. This enables your application to upload a file in the same way that a user would when using a form in a web browser.

File Management

The library can also perform some basic file management functions as well as send custom commands to the server. Some web servers also provide more advanced document management functions using WebDAV, an extension to HTTP for distributed document authoring.

HttpGetFileSize
Return the size of a file on the server without actually downloading the contents of the file. It is important to note that most servers will only return file size information for actual documents stored on the server, not for dynamically created content generated by scripts or web pages which use server-side includes.

HttpGetFileTime
Return the modification time for the specified file on the server. This can be used by your application to determine if the file has been changed since the time that you last uploaded or downloaded the contents.

HttpDeleteFile
Remove a file from the server. This operation requires that the current user have the appropriate permissions to delete the file. Not all servers support the use of this command, and it would typically require that the client authenticate prior to calling this function.

HttpCommand
This function enables the client to send any command directly to the server. This is commonly used to issue custom commands to servers that are configured to use extensions to the standard protocol.

Script Execution

The library also provides functions to execute server-side scripts, returning the output back to your application. Your program can pass additional data to the script, typically either as a query string or as form data, which is similar in concept to how arguments are passed to a command that is entered from the command prompt.

HttpGetData
In addition to being used to simply return the contents of a file, this function can also be used to execute a script on the server and return the output of that script to your program. Arguments to the script can be specified by passing them as a query string. For example, consider the following resource which is a PHP script:

```
/app/test.php?data1=value1&data2=value2
```

This would specify that the script /app/test.php is to be executed, and two arguments will be passed to that script: data1=value and data2=value2. The ampersand is used to separate the arguments, and they are grouped as pairs of values separated by an equal sign. Note that the actual format and value of the query string depends on how the script is written.

HttpPostData
An alternative method of providing information to a script is to post data to the script. Instead of the data being part of the resource name itself, posted data is sent separately and is provided as input to the script. This is the same method that is typically used when a user clicks the Submit button on a web-based form. This function requires the name of the script and the address of a buffer that

contains the data that will be posted. The resulting output from the script is returned to the caller in the same way that **HttpGetData** works.

HttpPostJson and HttpPostXml

These are specialized versions of the **HttpPostData** function which are designed to work with JSON and XML payloads. They will automatically set the correct content type and encoding for the request. If you are using the Unicode version of these functions, they will automatically convert the payload to standard UTF-8 encoding prior to submitting the data to the server. It is recommended you use these functions when accessing RESTful API services which expect JSON or XML requests.

HttpSubmitRequest

This is a high-level function which supports either a GET, PUT or POST request in a single function call. With this function, there is no need to explicitly connect to the server using **HttpConnect.** The entire transaction (i.e.: establishing the connection, submitting the request, processing the server response and then disconnecting) is handled automatically. If your application does not need to maintain a persistent connection to the server, using this function can eliminate a lot of boilerplate code which would otherwise be required.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol

The Internet Control Message Protocol (ICMP) library enables your application to send and receive ICMP echo datagrams. These are a special type of IP datagram which can be used to determine if a server is reachable, as well as determine the amount of time it takes for data to be exchanged with the local system. The ICMP library can also be used to trace the route that data takes from the local system to the server, which can be useful in determining why a connection to a particular system may be experiencing higher latency than normal.

The first step that your application must take is to initialize the library and then create a handle for the client session. Unlike many of the other libraries, there are no connection related functions because ICMP uses IP datagrams rather than TCP streams. The following functions are available for use by your application:

IcmpInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other ICMP API functions.

IcmpCreateHandle
This function will return a handle to a client session which is used in subsequent calls to the ICMP API. It is only required that you call this function if you're using the lower level ICMP functions to send and receive ICMP echo datagrams. The higher level functions like **IcmpEcho** and **IcmpTraceRoute** do not require a handle.

IcmpCloseHandle
Release the handle that was previously created by the call to **IcmpCreateHandle**. Any memory allocated by the library on behalf of the application is released and the datagram socket that was created is closed.

IcmpUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Ping and TraceRoute

To determine if a server is reachable, your application can send ICMP echo datagrams. You can also map the route between the local system and the server by sending a series of echo datagrams to each intermediate host. This is what the ping.exe and tracert.exe command line utilities do, and you can emulate that functionality in your own applications.

IcmpEcho
This is the simplest function you can use to send ICMP echo datagrams. Specify the server, the size of the ICMP datagram you want to send and the number of times you want to send it. The function will return if the operation was successful along with information such as the average number of milliseconds it took for the datagram to be returned by the server. Note that it is not required that you create a handle to use this function.

IcmpTraceRoute
This function will map the route that data packets take from your local system to a server. Whenever you send data over the Internet, that data is routed from one computer system to another until it reaches its destination. This function returns statistical information about each system that the data is routed through, and the latency between that system and the local host.

IcmpSendEcho
This is a lower level function that will send a single ICMP echo datagram. It can be used with applications that want to have more direct control over the process of how the datagrams are sent, and is typically used with asynchronous sessions.

IcmpRecvEcho

This is a lower level function that will receive a single ICMP echo reply datagram that was returned by the server. Typically this is used to receive datagrams that were sent in response to the **IcmpSendEcho** function. The **IcmpGetTripTime** function can be used after the function returns in order to check the amount of time it took to receive the datagram.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. The SocketTools IMAP library implements the current standard for this protocol, and provides functions to retrieve messages, create and manage mailboxes, and search for specific messages based on some user-defined search criteria.

The first step your application must take is to initialize the library, then establish a connection to the server and authenticate the client. The following functions are available for use by your application:

ImapInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other IMAP API functions.

ImapConnect
Establish a connection to the IMAP server. This function will return a handle to a client session which is used in subsequent calls to the IMAP API.

ImapDisconnect
Disconnect from the IMAP server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

ImapLogin
Authenticate yourself to the server using a username and password. This function should be called immediately after the connection has been established to the server.

ImapUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Managing Mailboxes

One of the primary differences between the IMAP and POP3 protocol is that IMAP is designed to manage messages on the mail server, rather than downloading all of the messages and storing them on the local system. To support this, IMAP allows the client to maintain multiple mailboxes on the server, which are similar in concept to message folders used by mail client software. A mailbox can contain messages, and in some cases a mailbox can contain other mailboxes, forming a hierarchy of mailboxes and messages, similar to directories and files in a filesystem. A special mailbox named INBOX contains new messages for the user, and additional mailboxes can be created, renamed and deleted as needed. Here are the most important functions for managing mailboxes:

ImapCheckMailbox
Check the mailbox for any new messages which may have arrived. Because messages are managed on the server, it is possible for new mail to arrive during the client session.

ImapCreateMailbox
Create a new mailbox on the server with the specified name.

ImapDeleteMailbox
Delete a mailbox from the server. Most servers will only permit a mailbox to be deleted if it does not contain any mailboxes itself. Unlike deleting a message, which can be undeleted, deleting a mailbox is permanent.

ImapExamineMailbox

Once the session has been established and authenticated, a mailbox should be selected. This enables the client to manage the messages in that mailbox. This function selects the specified mailbox in read-only mode so that messages can be read, but not modified. To select the mailbox in read-write mode, use the ImapSelectMailbox function.

ImapExpungeMailbox
Remove all of the messages marked for deletion and return updated status information about the current mailbox.

ImapGetFirstMailbox
This function begins the process of enumerating the available mailboxes on the server, according to certain criteria provided to the function.

ImapGetNextMailbox
This function returns the next mailbox from the server, based on the criteria specified to the **ImapGetFirstMailbox** function. When all the mailboxes have been returned, this function will return an error indicating that there are no more mailboxes.

ImapRenameMailbox
Renames an existing mailbox. One of the interesting uses of this function is the ability to rename the special INBOX mailbox. Instead of actually renaming it, it moves all of the messages to the new mailbox and empties the INBOX.

ImapSelectMailbox
Once the session has been established and authenticated, a mailbox should be selected. Selecting a mailbox enables the client to manage the messages in that mailbox. This function selects the specified mailbox in read-write mode so that changes can be made to the mailbox.

ImapUnselectMailbox
This function unselects the currently selected mailbox, and allows the caller to specify if messages marked for deletion should be expunged (removed) from the mailbox or reset back to an undeleted state.

Managing Messages

There are functions in the IMAP library for managing messages which enables the application to create, delete and move messages. To use these functions, a mailbox must be selected, either by calling the **ImapSelectMailbox** or **ImapExamineMailbox** function. Functions which modify the mailbox require that it be opened in read-write mode. Messages are identified by a number, starting with one for the first message in the mailbox.

ImapCopyMessage
Copy a message to a specific mailbox.

ImapDeleteMessage
Mark the specified message for deletion. Unlike the POP3 protocol, when a message is deleted on an IMAP server it can still be accessed. The message will not actually be removed from the mailbox unless the mailbox is expunged, unselected or the client disconnects from the server.

ImapEnumMessages
Return information about all, or a specific range of, messages on the server.

ImapGetMessageCount
Return the number of messages in the currently selected mailbox.

ImapGetMessageFlags
Return the status flags for a specific message. Messages have a number of flags which can be set that determines their status in the mailbox. For example, messages can be flagged as answered, seen

(read), recent (new) or deleted.

ImapGetMessageSize
Return the size of the message in bytes.

ImapSetMessageFlags
Set one or more flags for the specified message. Flags can be cleared, added or replaced using this function.

ImapUndeleteMessage
Remove the deletion flag from the specified message.

Viewing Messages

One of the more powerful features of the IMAP protocol is the ability to precisely select what kinds of message data you wish to retrieve from the server. It is possible to retrieve only specific headers, or specific sections of a multipart message. Because IMAP understands MIME formatted messages, it is possible to only retrieve the textual portion of a message without having to download any attachments that may have come with it.

ImapGetHeaderValue
This function returns the value for a specified header field in the message. Using this function, it is not necessary to download and parse the message header. To obtain the value of a header field in a specific part of a multipart message, use the **ImapGetHeaderValueEx** function.

ImapOpenMessage
This is a lower level function which opens a message for reading from the server. A more complex version of this function called **ImapOpenMessageEx** function allows you to specify the type of message data that you want, a specific part of a multipart message, and a byte offset into the message. The application would then call **ImapRead** to read the contents of the message, followed by **ImapCloseMessage** when all the message data has been read. Also see the **ImapGetMessage** function, which will return the contents of a message into a memory buffer.

ImapGetMessageParts
This function returns the number of parts in a multipart message and is useful for determining if a message is a simple message or a MIME formatted message with multiple parts, such as one that includes file attachments.

Downloading Messages

In some cases, it may be preferable to download a complete message from the server to the local system. This can be easily done with a single function call.

ImapGetMessage
This function retrieves the specified message and stores it in a buffer provided by the caller; you can specify the type of message data that you want, a specific part of a multipart message and the amount of data that you want. For example, it is possible to request that only the first 1500 bytes of the body of the 3rd part of a multipart message should be returned.

ImapStoreMessage
This function downloads a complete message and stores it as a text file on the local system.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Client

The Large Language Model (LLM) client API enables your application to interact with modern AI services to generate text, answer questions and process natural language input. Rather than requiring your application to work directly with a specific provider, the API provides a common interface that can be used with both cloud-based services and locally hosted models.

The library handles the provider-specific details of creating requests, managing connections and returning responses to your application. This abstraction allows you to focus on how your application uses the generated content, rather than the specifics of how each provider implements their API. It also makes it possible to develop and test using one provider, and then switch to another with minimal changes to your code.

The API is session based, allowing your application to send individual prompts or maintain a conversation by preserving message history across multiple requests. Each session represents a connection to a provider and can be configured with options such as the model, timeouts and generation parameters.

A number of cloud-based and local providers are supported. For an overview of currently available services and their capabilities, please refer to the providers page. If you are new to development with large language models, the glossary of terms provides definitions for common concepts used throughout this documentation.

Getting Started

Before sending requests, your application must establish a connection to a supported provider. This includes specifying the provider type, the model to use, and any required authentication credentials such as an API key.

The library supports a simplified connection model for most applications, as well as an extended session structure that provides more control over options such as timeouts, temperature, token limits, and other generation parameters. Once a connection is established, a client handle is returned which is used in subsequent calls. This handle represents the connection between your application and a provider.

LlmInitialize
Initialize the library for the current process. This must be the first function call that the application makes before calling the other LLM API functions.

LlmConnect
Connect to a provider using a simplified set of parameters. This function specifies the provider, options, timeout period, base URL, model name and API key. It returns a client handle which is used in subsequent calls to the library.

LlmConnectEx
Connect to a provider using an extended session structure. This function provides additional control over options such as the proxy URL, API version, application name, organization ID, system prompt, temperature, reasoning level and token limits.

LlmDisconnect
Disconnect from the provider and release the memory allocated for that client session. After this function is called, the client handle is no longer valid.

LlmUninitialize
Release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Sessions and Connections

A client session represents the connection between your application and a provider. The session stores information such as the selected provider, model name, authentication credentials, timeout period and other options that control how requests are processed.

LlmCreateSession
Initialize a session structure using common connection parameters. This can be useful when your application wants to prepare a session before calling the extended connection function.

LlmCreateSessionEx
Initialize a session structure using additional parameters for advanced configuration. This function is useful when your application needs more control over how the provider connection is established.

LlmDeleteSession
Release the memory allocated for a session structure created by the library. This function should only be used with session structures allocated by the LLM API.

LlmIsConnected
Determine if the specified client handle represents an active connection to a provider.

LlmResetSession
Reset the current session state while preserving the client handle. This can be used when your application wants to begin a new conversation without creating a new connection.

Prompts and Responses

The primary purpose of the library is to send input to a language model and receive generated output. Your application provides a prompt, or message, which is represented as an LLM_MESSAGE structure. The library formats the request according to the requirements of the selected provider and model. The response returned to your application is represented as an LLM_RESPONSE structure. It includes the generated text and related status information. The combination of a prompt and response is called a message "turn" and is stored in the session's conversation history.

In the API documentation, you'll see references to both ***prompts*** and ***messages***. A ***prompt*** refers to the text sent to the model containing a request or instructions. The API represents this input as a ***message***, which includes the prompt text along with additional metadata such as the role and other request options. In most cases, these terms can be used interchangeably.

The library formats the request according to the requirements of the selected provider and model, sends it to the service, and returns the generated response. This response typically includes the generated text, status information, and optional metadata such as token usage.

LlmSendMessage
Send a prompt to the model and return the generated response. This function accepts a string prompt and returns the response in a null-terminated string buffer. It doesn't require the use of the LLM_MESSAGE and LLM_RESPONSE structures. This function is appropriate for most applications that simply want to provide a prompt and receive a text response.

LlmSendMessageEx
Send a message using the LLM_MESSAGE structure and receive the response in an LLM_RESPONSE structure. This function provides additional information about the request and can return more detailed response metadata.

LlmCancelRequest
Cancel a request that is currently in progress. This can be used to keep your application responsive when a provider takes too long to return a response.

LlmIsBlocking
Determine if the client session is currently waiting for a request to complete.

Conversation Management

The library supports multi-turn conversations by allowing your application to maintain a list of messages that represent the current context. Each message is assigned a role, such as user, assistant, or system, which helps

guide how the model interprets the input. By including previous messages in subsequent requests, your application can create more natural and context-aware interactions. The library provides helper functions to manage this message history, making it easier to build conversational workflows without manually tracking every exchange.

LlmGetFirstMessage
Return the first message stored in the conversation history. This is the first step in enumerating the messages maintained by the session.

LlmGetNextMessage
Return the next message stored in the conversation history. This function is called repeatedly until it indicates that all messages have been returned.

LlmFindMessage
Search the conversation history for a message that contains the specified text.

LlmTrimHistory
Remove older messages from the conversation history.

LlmClearHistory
Remove all messages from the current conversation history.

LlmGetHistorySize
Return the maximum number of messages that will be retained in the conversation history.

LlmSetHistorySize
Specify the maximum number of messages that should be retained in the conversation history.

Providers and Models

The library provides functions to enumerate available providers and models, as well as retrieve detailed information about their capabilities. This includes supported features, token limits, and other constraints that may affect how requests are processed. This allows your application to dynamically adapt to different environments, select appropriate models, or present options to the end user.

LlmEnumProviders
Enumerate the providers supported by the library and return information about each provider.

LlmGetProviderInfo
Return information about a specific provider, including its name, host name, default model and capability flags.

LlmGetProviderName
Return the display name for a provider.

LlmGetProviderFlags
Return the capability flags associated with a provider.

LlmEnumModels
Enumerate the models available from the specified provider. The list can be filtered to include models with specific capabilities.

LlmGetModelInfo
Return detailed information about a model, including its token limits, capabilities and status flags.

LlmValidateModel
Determine if a model name is valid for the specified provider.

Generation Options

The response generated by a model can be influenced by several options. These include the system prompt,

timeout period, maximum number of tokens, temperature and reasoning level. In most cases, the default values are appropriate, but these functions can be used when your application needs to customize how responses are generated.

Some models also support advanced features such as structured output, reasoning modes, or multimodal input. The API exposes these capabilities in a consistent way, allowing your application to take advantage of them when available.

LlmGetSystemPrompt
Return the system prompt associated with the client session. The system prompt provides instructions that help define how the model should respond.

LlmSetSystemPrompt
Specify the system prompt that should be used with subsequent requests.

LlmGetTimeout
Return the current timeout period for the client session.

LlmSetTimeout
Specify the timeout period used when waiting for a provider to return a response.

LlmGetModelContextSize
Return the context size for the specified model. This value can help your application determine how much conversation history can be sent with a request.

LlmSetModelFlags
Modify the capability flags associated with a model. This can be useful when your application needs to override the default behavior for a specific provider or model.

Token Usage and Limits

Language models process text as tokens. The number of tokens used by a request can affect whether the request fits within the model context window and may also affect usage costs for cloud-based providers. The library provides functions to estimate token counts and retrieve usage information returned by the provider.

LlmEstimateTokenCount
Estimate the number of tokens used by a string. This is intended as a general estimate and should not be treated as an exact count for all providers and models.

LlmGetTokenUsage
Return token usage information for the client session, including input and output token counts when that information is available from the provider.

Diagnostics and Utility Functions

The library includes several functions that can be useful for diagnostics, logging and general application support. These functions can help your application report errors, trace requests, manage cache behavior and format timestamps returned by the API.

LlmGetLastError
Return the last error code reported by the library for the current thread.

LlmGetErrorString
Return a string that describes the specified error code.

LlmEnableTrace
Enable diagnostic tracing for the current process or client session. This can be useful when troubleshooting provider requests and responses.

LlmDisableTrace

Disable diagnostic tracing.

LlmGetUserAgent
Return the user agent string used with provider requests.

LlmSetUserAgent
Specify the user agent string used with provider requests.

LlmGetCacheTime
Return the amount of time that provider and model information may be cached by the library.

LlmSetCacheTime
Specify the amount of time that provider and model information may be cached by the library.

LlmFormatTimestamp
Convert a timestamp value to a formatted string.

LlmConvertTimestamp
Convert a timestamp value to a Windows system time structure.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Mail Message

The Mail Message library can be used to create and process messages in the format defined by the Multipurpose Internet Mail Extensions (MIME) standard. When a message is parsed, it is broken into parts, each consisting of two sections. The first part is called the header section and it describes the format of the data and how it should be represented to the user. The second section is the data itself. A typical mail message without file attachments has one part, with the body of the message being the data. Messages with attachments have multiple parts, each with a header describing the type of data. The library can be then used to extract the data from a multipart message and save it to a file on the local system, delete the part from the message, or add additional parts to the message, such as attaching a file.

The library can also be used to create new multipart messages with alternative content, such a message with both plain text and styled HTML text. Once a message has been created, files can be attached to the message and the application can make any other changes that are needed. The library provides complete access to all headers and content in a multipart message, including the ability to create your own custom headers and make modifications to specific sections.

The first step that your application must take is to initialize the library, then establish a connection to the server and authenticate the client. The following functions are available for use by your application:

MimeInitialize
Initialize the library for the current process. This must be the first function call that the application makes before calling the other MIME API functions.

MimeComposeMessage
Compose a new message using the specified header field values and content. Using this function, you can create a message with the From, To, Cc and Subject headers already defined, along with any text for the message. You can also optionally provide both plain and styled HTML text versions of the message and the function will automatically create a multipart message. The function returns a handle which can be used by the application to make further modifications, such as attaching files to the message.

MimeCreateMessage
Create a new, empty message and return a handle which can be used by the application to reference the message. In most cases, using the MimeComposeMessage function is preferred, however there may be some situations where the application needs a message handle when the contents of the message aren't known at the time. For example, parts of a message may be created dynamically based on the results of several database queries.

MimeOpenMessage
Opens a text file which contains a MIME structured message. The entire contents of the file are read into memory and parsed. If successful, a handle is returned which can be used by other functions to access specific parts of the message, examine and change header values and extract file attachments. This function cannot be used to open messages which are stored in a proprietary binary format.

MimeCloseMessage
Releases the memory allocated for the message and closes the message handle. After this function returns, the handle is no longer valid. This function should be called when the message is no longer needed by the application.

MimeUninitialize
Release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Message Headers

Each message has one or more headers fields which provide information about the contents of the message. For example, the "From" header field specifies the email address of the person who sent the message. There are a fairly large number of header fields defined by the MIME standard, and applications can also create their own custom headers if they wish. The library gives the application complete access to the header fields in a message. Headers can be examined, modified, created or removed from the message as needed.

MimeGetMessageHeader
This function returns a pointer to a null-terminated string which contains the value of the specified header field in the current message part. For languages like Visual Basic, it is recommended that you use the **MimeGetMessageHeaderEx** function instead.

MimeGetMessageHeaderEx
This function copies the value of a header field into a null terminated string buffer that you provide. This function also allows you to specify the message part in a multipart message. To return the value of the common header fields such as "From", "To" and "Subject", you should specify a message part of zero.

MimeGetFirstMessageHeader
This function returns the value of the first header defined in the current message part, copying it into the string buffer that you provide. This is used in conjunction with the MimeGetNextMessageHeader function to enumerate all of the headers that have been defined.

MimeGetNextMessageHeader
This function returns the value of the next header defined in the current message part. It should be called in a loop until it returns a value of zero (False) which indicates that the last message header has been returned.

MimeSetMessageHeader
Set a message header field to the specified value in the current message part. If the value is a null pointer or empty string, the message header will be deleted from the message.

MimeSetMessageHeaderEx
Set a message header field to the specified value. This version of the function allows you to also specify the message part, rather than just using the current message part. To return the value of the common header fields such as "From", "To" and "Subject", you should specify a message part of zero. If the value is a null pointer or empty string, the message header will be deleted from the message.

MimeDeleteMessageHeader
Delete the specified message header from the current message part.

Message Contents

The content or body of a message contains the text that is to be read or processed by the recipient. It may be a simple, plain text message or it may be more complex, such as a combination of plain and styled HTML text or the data for a file attachment. The library provides complete access to the contents of the message, enabling the application to modify, extract, replace or delete specific sections of the message.

MimeGetMessageText
Copy the text from the body of the current message part into a null terminated string buffer. The offset and length of the text being copied can be specified, which enables you to return just a portion of the message. For example, it would be possible to return just the first 1K byte of a text message.

MimeSetMessageText
Set or replaces the text in the body of the current message part. You can specify an offset in this function, enabling you to replace just portions of the message.

MimeAppendMessageText

Append text to the body of the current message part. Instead of replacing existing text, the new text will be added to the end of the message.

Multipart Messages

Most typical messages contain a single part, which consists of the message headers followed by the contents of the message. However, when files are attached to a message or alternative content types such as HTML are used, a more complex multipart message is required. With a multipart message, the contents of the message are split into logical sections with each section containing a specific part of the message. For example, when a file is attached to a message, one part of the message contains the text to be read by the recipient and another part contains the data for the file.

The first of a multipart message is called part 0, and contains the main header block. This is what defines the headers that you are most familiar with, such as "From", "To" and "Subject". The body of this message part is typically a plain text message that indicates that this is a multipart message. This is done for the benefit of older mail clients that cannot parse MIME messages correctly. Next part, part 1, typically contains the actual body of the message that would be displayed by the mail client. Additional parts may contain file attachments and other information. In the case of a multipart message that contains both plain and styled HTML text versions of a message, part 1 is typically the plain text version of the message while part 2 contains the HTML version. The mail client can then make a decision based on its own configuration as to which version of the message it displays.

MimeGetMessagePartCount
Return the number of parts in the message. A simple message will contain just one part. This can be used to determine if the message is a MIME multipart message, with a value greater than one indicating that it is.

MimeGetMessagePart
Return the current message part. Each message has the concept of a "current message part", which is the default message part that the other functions will use. When a message is first created, the current message part is zero. The simplest way to think of message parts is as a zero-based index into an array into sections of the message.

MimeSetMessagePart
Change the current message part. Once the current message part has been set, it will remain as the current part until explicitly changed or until a new message part is created.

MimeCreateMessagePart
Create a new, empty message part. If the message was not originally a multipart message, it will be restructured into one. Otherwise, the new part is simply added to the end of the message. This function will cause the current message part to change to the new part that was just created.

MimeCreateMessagePartEx
Create a new message part using the specified character set and text. If the message was not originally a multipart message, it will be restructured into one. Otherwise, the new part is simply added to the end of the message. This function will cause the current message part to change to the new part that was just created.

MimeDeleteMessagePart
Delete the message part from the message. If the message part is in the middle of the message, it will cause the subsequent parts of the message to be reordered. You should not delete part zero to delete a message; use the **MimeDeleteMessage** function instead.

Importing Messages

The library can be used to import existing messages, either from memory or from a file. Once the message has been parsed, the application can examine or modify specific parts of the message. The following functions are

provided to import the contents of a message:

MimeImportMessage
The simplest method of importing a message, this function reads the contents of the specified file and imports it into the current message. This function is typically called immediately after MimeCreateMessage to load a file into a new message context.

MimeImportMessageEx
This version of the function enables you to import the message from a file, the system clipboard, a pointer to a memory buffer or a global memory handle. This is typically used to import messages that were retrieved using the POP3 or IMAP protocols by passing a pointer to the buffer that contains the data returned by the server.

Exporting Messages

After a message has been created or modified, it can be exported to a file or to memory. Exporting the message to a memory buffer is particularly useful when using the library with another one of the SocketTools libraries. For example, the contents of a message can be exported to memory, and that memory address can be passed to the Simple Mail Transfer Protocol (SMTP) library for delivery to the recipient. The following functions are provided to export the contents of a message:

MimeExportMessage
This function exports the current message to a file. When using this function, only certain headers are exported and they may be reordered. To force all headers to be included in the message or to preserve the order of the headers, use the extended version of this function.

MimeExportMessageEx
This extended version of the function exports the message to a file, the system clipboard, a memory buffer or returns a handle to a global memory buffer that contains the message. There are additional options that enable you to control what headers are exported, and whether the order the headers were set should be preserved.

File Attachments

In addition to simple text messages, one or more files can be attached to a message. The process of attaching a file involves creating a multipart message, encoding the contents of the file and then including that encoded data in the message. The following functions are provided to manage files attached to the message, as well as attach files to an existing message:

MimeAttachFile
This function attaches the contents of the file to the message. The file will be attached using the specified encoding algorithm and will become the current message part. If the message is not a multipart message, it will be converted to one; if it already is a multipart message, the attachment will be added to the end of the message.

MimeAttachData
This function works in similar fashion to **MimeAttachFile**, except that instead of the contents of a file, the data in a memory buffer will be attached to the message. If the message is not a multipart message, it will be converted to one; if it already is a multipart message, the attachment will be added to the end of the message.

MimeAttachImage
This function attaches an inline image file to the message. It is similar to the **MimeAttachFile** function, except that the image is designed to be referenced as an embedded graphic in an HTML message. This function will automatically set the correct header values for an inline image attachment, and enables the developer to specify a content ID which is used in the HTML message.

MimeGetAttachedFileName

Return the name of a file attachment in the current message part. This function serves two purposes, to determine if the current message part contains a file attachment, and if so, what file name should be used when extracting that attachment.

MimeExtractFile

Extract the file attachment in the current message part, storing the contents in a file. The attachment will automatically be decoded if necessary. This function also recognizes uuencoded attachments that are embedded directly in the body of the message, rather than using the standard MIME format.

Mail Addresses

The library has functions which are designed to make it easier to work with email addresses. Addresses are typically in the format of "user@domain.com" however additional information can be included with the address, such as the user's name or other comments that aren't part of the address itself. The library can parse these addresses for you, returning them in a format that is suitable for use with other protocols such as the SMTP library.

MimeParseAddress

Parse an email address that may include an address without a domain name or comments in the address, such as the user's name. For example, the From header field may return an address like "Joe Smith <joe@example.com>"; this function would parse the address and return "joe@example.com", the actual address for the user.

MimeEnumMessageRecipients

It is common for certain headers to contain multiple addresses separated by a comma. These addresses may also include comments such as the user's name. This function parses a string and returns a list of valid addresses. For example, the To header field may contain "Tom Jones <tom@example.com>, Jerry Lewis <jerry@example.com>"; this function would return "tom@example.com" and "jerry@example.com" as the two addresses listed.

Message Storage

The library has a collection of functions which makes it simple for an application to store a group of messages together in a single file, search for and retrieve specific message. The collection of messages is referred to as a "message store" and messages may either be stored in a plaintext format or in a compressed binary format.

MimeOpenMessageStore

This function is used to open an existing message store or create a new one. The function returns a handle which is used to access the messages in the storage file, and must be closed using the **MimeCloseMessageStore** function. The storage files may either be plaintext, or stored in a compressed format. It also supports opening storage files in the UNIX mbox format.

MimeGetStoredMessageCount

This function returns the number of messages that currently in the message store. Each message is referred to by an integer which is its index into the storage file. The first stored message has a value of 1, and increments with each additional message in the storage file.

MimeFindStoredMessage

An application can search the message store for messages that match any header value. Searches can be complete or partial, and may be case-sensitive or case-insensitive. For example, this function can be used to enumerate all of the messages in the storage file that were sent by a specific user or match a specific subject.

MimeGetStoredMessage

This function returns a handle to the message in the storage file, and can be used with any of the other functions in the Mail Message API. The application can also request its own private copy of the

message, which it can modify independently of what is currently in the message store.

MimeStoreMessage

This function stores a new message, specified by its message handle, to the open message store. Note that the message store must be opened for write access, and this function will always append the message to the storage file. The new message index is returned to the caller.

MimeDeleteStoredMessage

This function flags a message for deletion from the message store. Once a message has been flagged for deletion, it may no longer be accessed by the application. When the storage file is closed, the contents of the deleted message will be removed from the file.

MimeReplaceStoredMessage

This function replaces an existing message in the storage file, overwriting it with the specified message. Unlike many of the other functions which do not permit the application to reference a deleted message, this function can be used to replace a previously deleted message.

MimeCloseMessageStore

The message store must be closed when the application has finished accessing it. The storage file is updated with any changes, all deleted messages are purged and the handle to the open file is closed. If the storage file is locked for exclusive access, this function will release that lock, allowing another process to open the file.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) library enables applications to access a news server, list the available newsgroups, retrieve articles and post new articles. It is common for this library to be used in conjunction with the Mail Message library to construct the articles, since a news article uses the same general format as an email message.

The first step that your application must take is to initialize the library, then establish a connection to the server and authenticate the client. The following functions are available for use by your application:

NntpInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other NNTP API functions.

NntpConnect
Establish a connection to the NNTP server. This function will return a handle to a client session which is used in subsequent calls to the NNTP API.

NntpAuthenticate
Provide a user name and password to authenticate the client session. This should only be used if required by the server. Not all news servers require authentication, and some only require authentication when posting articles. If you attempt to perform a function that requires authentication, an error will be returned that indicates you should authenticate and then retry the operation.

NntpDisconnect
Disconnect from the NNTP server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

NntpUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Newsgroups

News articles are posted in hierarchical groups, similar to how files are stored in folders. Each level in the newsgroup hierarchy is separated by a period, so newsgroup names look like microsoft.public.vc. This is Microsoft's newsgroup for articles about Visual C++ programming. Additional subgroups are used to further narrow the topic; for example, there's the microsoft.public.vc.3rdparty newsgroup for third party tools and components for Visual C++, and the microsoft.public.vc.atl newsgroup which discusses issues related to the Active Template Library. The NNTP API provides the following functions for accessing newsgroups on the server:

NntpListGroups
This function requests that the server return a list of all of the newsgroups that are available. If the function is successful, the application should call the NntpGetFirstGroup function to begin processing the group list.

NntpListNewGroups
This function is similar to the **NntpListGroups** function in that it requests the server to return a list of available newsgroups. However, the application can request that only groups which were created since a specific date should be returned. This allows the application to maintain a list of newsgroups on the local system, and then use this function to periodically update that list based on the date it was last modified.

NntpGetFirstGroup
This function is used in conjunction with the **NntpListGroups** or **NntpListNewGroups** functions to

enumerate the newsgroups that are available on the server. Information about the newsgroup is returned in a structure, including the name of the group, the first available article number and the last available article in the group.

NntpGetNextGroup
This function is used to return information about the next newsgroup in the list. It should be called in a loop until it returns zero (False).

NntpSelectGroup
This function is used to select a newsgroup as the current group. Once selected, the application has access to the articles in that newsgroup.

NntpGetGroupName
Return the name of the currently selected newsgroup.

NntpGetGroupTitle
Return a description of the currently selected newsgroup. Note that not all newsgroups have associated descriptions, and some servers may not support the extended command which is used to retrieve the description.

News Articles

News articles are the messages posted to one or more newsgroups. Articles are referenced by their article number, which is a value assigned by the news server. These articles have a structure that is the same as an email message, with some slightly different headers. Because of this, you can use the Mail Message API to parse articles that you retrieve, as well as create new articles to post to the server. The following functions are used to access and create news articles:

NntpListArticles
This function requests that the server return a list of articles that are available in the current newsgroup. The application can request that all articles be returned, or only those articles which fall into a certain range of article numbers.

NntpGetFirstArticle
This is the first function that should be called after the **NntpListArticles** function. It will return information about the first article in the list. Article information is returned in a structure which includes information such as the article ID, size, subject, author and date that the article was posted.

NntpGetNextArticle
This function returns the information about the next article in the list. It should be called in a loop until the function returns zero (False).

NntpGetArticleRange
Return the range of articles that are available in the currently selected newsgroup. These are the first and last valid article numbers that can be used to retrieve an article from the server. It is important to keep in mind that there is no requirement that articles be stored contiguously with no gaps in between them. For example, say the first available article number in the newsgroup is 101 and the last available article number is 120; it does not necessarily mean that there are 20 available articles. Articles 112 and 118 may have been removed, in which case your application would get an error when trying to access them. The inability to access an article within the article range should not be considered a fatal error; the program should simply move on to the next message.

NntpGetArticle
Retrieve an article from the server, storing the contents in memory. This can be used to process the contents of an article without the overhead of storing it in a file on the local system.

NntpStoreArticle

Retrieve an article from the server and store it in a file on the local system.

NntpPostArticle
This function posts an article to one or more newsgroups on the server. A newsgroup article is similar to an email message, and the MIME API may be used to create the article headers and body. One important difference is that the message must contain a header named "Newsgroups" with the value set to the newsgroup or newsgroups that the article should be posted to; multiple newsgroups should be separated by commas. If this header is not defined, the posting will be rejected by the server and the function will return an error. You should also be aware that some servers limit the number of newsgroups that a message can be posted to. When an article is posted to more than one newsgroup at a time, this is called cross-posting. Current convention says that an article should not be cross-posted to more than five newsgroups at a time. Also keep in mind that multi-posting (posting the same article to different newsgroups separately) is generally discouraged and should never be done on USENET.

Attaching Files

It is possible to attach files to newsgroup articles; however it should only be done if it is considered appropriate for the group. Many newsgroups have their own acceptable use policies which determine whether or not file attachments, particularly large binary files, are acceptable. If the newsgroup accepts attachments, you can use one of several methods for posting files. It is recommended that you use the File Encoding API to handle the actual encoding of the data.

Uuencode
A uuencoded file attachment is included directly in the body of the message. Because the MIME API creates a multipart message even when uuencoding is specified, the File Encoding API should be used to encode the data and then it should be included in the main body of the message.

Base64
A Base64 file attachment has the same structure as what is used by email messages. This requires that a multipart message be created, with the encoded data attached as a part of the message. You can use the MIME API to create this kind of message. Note that not all third-party newsreaders correctly handle multipart messages.

yEnc
A popular encoding method used on USENET is called yEnc. Similar to uuencoded attachments, the file data is part of the body of the message. The File Encoding API should be used to encode the data and then it should be included in the main body of the message.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol

The Network Time protocol library enables an application to retrieve the current time from a server, and optionally synchronize the local system time using that value. The first step that your application must take is to initialize the library. After the library has been initialized, the application can request the current time from a system and update the local system clock if necessary.

TimeInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other Time API functions.

GetNetworkTime
Return the current time from a server. The time is expressed as a 32-bit integer value which represents the number of seconds since midnight, 1 January 1900 UTC.

UpdateLocalTime
Update the local system time with the value returned by **GetNetworkTime**. This function requires that the current user have the appropriate permissions to modify the system time or the function will fail.

TimeUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Time Conversion

Windows applications typically use a structure called SYSTEMTIME to represent date and time values. The library has two functions which will enable you to convert between the value returned by **GetNetworkTime** and the SYSTEMTIME structure.

ConvertNetworkTime
This function will convert the value returned by **GetNetworkTime** into a SYSTEMTIME structure, adjusting for the local timezone if required.

ConvertSystemTime
This function will convert a SYSTEMTIME structure into a 32-bit integer value. This value may be passed to the **UpdateLocalTime** function to update the local system clock.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Library

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

News feeds can be accessed remotely from a web server, or locally as an XML formatted text file. The source of the feed is determined by the URI scheme that is specified. If the http or https scheme is specified, then the feed is retrieved from a web server. If the file scheme is used, the feed is considered to be local and is accessed from the disk or local network. The News Feed library provides an API that enables you to open a feed by URL and iterate through each of the items in the feed or search for a specific feed item. The API also provides a function that can be used to parse a string that contains XML data in RSS format, where the feed may have been retrieved from other sources such as a database.

The first step your application must take is to initialize the library, which will load the required system libraries and initialize the internal data structures that are used. You must call the initialization function before attempting to call any other function in the library.

RssInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other SMS API functions.

RssUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

News Channels

A news feed consists of a channel that contains each of the news feed items. The news channel is represented in the library by an **HCHANNEL** handle. This handle is used with the other functions to reference the specific news feed channel. Information about the news feed, such as the title of the channel and the date it was last modified, is returned in an RSSCHANNEL structure.

RssOpenFeed
Open the channel by specifying a URL to the resource that contains the news. The URL can identify a remote feed that is downloaded using the HTTP or HTTPS protocols, or it can be a file on the local system or network.

RssParseFeed
Parse a string buffer that contains a news feed. This function is similar to the **RssOpenFeed** function, however it used to parse a string that contains the news feed. This function would typically be used when the feed content is obtained from a different source, such as a database or by using a different protocol. For example, the news feed could be downloaded using the FTP API and then passed to this function.

RssCloseFeed
This function closes the handle that was allocated by a previous call to the **RssOpenFeed** or **RssParseFeed** function. When the information in a news feed is no longer needed, this function must be called to release the resources allocated to process the feed. After this function is called, the handle is no longer valid.

RssValidateFeed
This function is used to validate the contents of a news feed, ensuring that it is structured correctly.

The function will return information about the feed, including the number of news items in the feed and the date that it was last modified. The news feed can be specified either as a URL to a remote resource or as a file on the local system or network.

RssStoreFeed

This function is used to store a news feed as a file on the local system. This is typically used to cache the contents of a news feed or to track the changes made to the feed over time. It is recommended that the application periodically check the publication date of the feed to ensure that they have current version.

News Items

News feed items are identified by a numeric value called the item ID. This is used with other functions to return information about a specific news item. The first item in a news feed has an ID of one and it increments for each additional item in the feed, although it is recommended that applications treat this an opaque value. Functions are provided which will return the number of items in a news feed, information about each item and enable you to easily enumerate each of the items in the feed. Information about a particular news item is returned in an RSSCHANNELITEM structure.

RssGetItemCount

This function will return the number of news items in the feed.

RssGetFirstItem / RssGetNextItem

These functions are used to enumerate the items in the news feed. Details about each news item is returned in an **RSSCHANNELITEM** structure which contains information such as the title, description and author of the news item.

RssGetItem

This function is used to return information about a specific news item based on the item ID. When the function returns, it will populate an **RSSCHANNELITEM** structure. Although this function can be used to effectively enumerate all of the news feed items by starting with an item ID value of one, it's recommended that you use the **RssGetFirstItem** and **RssGetNextItem** functions instead. Your application should never make an assumption about the actual value of the item ID because there's no guarantee that future versions of the library will assign item IDs sequentially. Best practices dictate that the **RssGetItem** function should only be called using an item ID that was previously obtained by a call to either the **RssGetFirstItem**, **RssGetNextItem** or **RssFindItem** functions.

RssFindItem

This function is used to search the feed channel for a specific item, based either on its GUID, title, link or publication date. When searching for a specific item, only searches by GUID are guaranteed to return a unique news item. However, since not all news feeds may provide GUIDs for their news items, additional search criteria can be used when necessary. If the function is successful, it will populate an **RSSCHANNELITEM** structure with information about the news feed item.

RssGetItemText

This function is used to return a copy of the news item description based on the item ID. Internally, this function calls **RssGetItem** and then copies the item description to the string buffer that is provided by the caller. Note that it is the responsibility of the application to display the text in the appropriate format. Most news feeds will either use plain text or HTML formatted text for the item description.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol

The Post Office Protocol (POP3) library enables an application to retrieve a user's mail messages and store them on the local system. The POP3 API provides support for all of the standard functionality such as listing and downloading messages, as well as extended features such as the ability to retrieve only the headers for a message or just specific header values. The library also has functions for changing the user's password and sending messages if they are supported by the server.

The first step your application must take is to initialize the library, then establish a connection to the server and authenticate the client. The following functions are available for use by your application:

PopInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call the application makes before calling the other POP3 API functions.

PopConnect
Establish a connection to the POP3 server. This function will return a handle to a client session which is used in subsequent calls to the POP3 API.

PopLogin
Authenticate yourself to the server using a username and password. This function should be called immediately after the connection has been established to the server. You can specify either the standard authentication method, or the APOP authentication method if required by the server.

PopDisconnect
Disconnect from the POP3 server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

PopUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Managing Messages

There are functions in the POP3 library for managing messages which enables the application to list, delete and retrieve messages stored on the server. Messages are identified by a number, starting with one for the first message in the mailbox. The most typical operation for a POP3 client is to retrieve each message, store it on the local system and then delete the message from the server. Any processing that is done on the message would then be done on the local copy.

PopGetMessageCount
Return the number of messages available for retrieval. There are two values the application should use. One is the number of currently available messages and the other is the last valid message number. As messages are deleted from the server, the total number of available messages will decrease; however, the last available message number will remain constant.

PopGetMessageCountEx
An extended version of the **PopGetMessageCount** function, this function will return the number of available messages, along with the last available message number and the total size of all messages in the mailbox.

PopGetMessage
Retrieve a message from the server, storing the contents in memory. This can be used to process the contents of a message without the overhead of storing it in a file on the local system.

PopStoreMessage
This function downloads a complete message and stores it as a text file on the local system.

PopDeleteMessage

Mark the message for deletion. When the connection with the server is closed, the message will be removed from the user's inbox. An important difference between the POP3 and IMAP protocols is that when a message is marked as deleted on a POP3 server, that message can no longer be accessed. An attempt to retrieve a message after it has been marked for deletion will result in an error. The only way to undelete a message once it has been deleted is to terminate the connection with the server by calling the **PopReset** function instead of calling **PopDisconnect**.

PopGetMessageSize

This function returns the size of the message in bytes. One thing to be aware of when using this function is that some servers will only return approximate message sizes. In addition, because of the difference between the end-of-line characters on UNIX and Windows systems, the size reported by the server may not be the actual size of the message when stored on the local system. Therefore, the application should not depend on this value as an absolute. For example, it should not use this value to determine the maximum number of bytes to read from the server; instead, it should read until the server indicates that the end of the message has been reached.

Message Headers

The POP3 API also includes functions which enable the application to access just the headers for a message. This can be useful if the program doesn't want to incur the overhead of downloading the entire message contents. The following functions can be used to examine the headers in a message:

PopGetMessageHeaders

This function returns the complete set of headers for the specified message. If your program has to process multiple header fields, this is the most efficient method to use. It is possible to retrieve specific header values, however not all servers support that option and it is somewhat slower because it involves sending individual commands to request each value.

PopGetHeaderValue

This function returns the value for a specific header field in a message. This function does not require that you parse the message headers; however it does incur additional overhead. It is also important to note that not all servers support the command that is used to request the header value. If this function fails with the error that the feature is not supported, you should use the **PopGetMessageHeaders** function instead.

PopGetMessageId

This function returns the value of the Message-ID header in the specified message. This is a unique string that is used to identify the message. Note that it is not the same as the UID value returned by the POP3 server.

PopGetMessageUid

This function returns the unique ID (UID) that the server has associated with the message. The UID can be used by an application to track whether or not it has previously viewed the message. Unlike the message number, which can change between client sessions, the message UID is guaranteed to be the same value across sessions until the message is deleted.

PopGetMessageSender

This function returns the email address of the person who sent the message. This function requires that the server support extended POP3 commands. If the server does not support the command used to retrieve the sender, it will return a value of zero. Applications should not depend on this function returning a valid address. Typically it is used for informational purposes, such as displaying the sender to the user as a message is being retrieved.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Remote Command Protocol

The Remote Command protocol enables an application to execute commands on a server, with the output of the command returned to the client. The SocketTools library actually implements three related protocols: rexec, rshell and rlogin. The choice of protocols is determined by the port that is selected when a connection is established.

Rexec
The rexec protocol enables a client application to execute a command on a server. Output from the command is returned to the client and the connection is closed when the command terminates. The client connects on port 512 and must provide a user name and password to authenticate the session.

Rshell
The rshell protocol is similar to rexec in that it enables a client to execute a command on a server. Output from the command is returned to the client and the connection is closed when the command terminates. The client connects on port 514 and must provide a user name. The primary difference between the rexec and rshell protocols is that rshell does not require a password. Instead, it uses what is called "host equivalence" to determine if the client is permitted to execute commands as that user. On a UNIX based operating system, host equivalence is controlled by the /etc/hosts.equiv and the .rhosts file in the user's home directory. These files list the host names and user names which are permitted to execute commands using the rshell protocol. Consult your operating system manual pages for more information about how to configure host equivalence.

Rlogin
The rlogin protocol is similar to Telnet in that it provides an interactive terminal session. The connection is closed when the user logs out or the shell process on the server is terminated. The client connects on port 513 and must provide a user name and terminal type. If there is an entry in the host equivalence tables for the user and local host, then the client will be automatically logged in and provided with a shell prompt. If there is no host equivalence, the client will be prompted for a password. The terminal emulation library can be used to provide ANSI or DEC VT-220 emulation services if needed.

An important consideration when deciding whether to use rexec, rshell or rlogin is how the server is configured and the type of command being executed. If there is no entry for the local host in the server's host equivalence tables, then the rexec command should be used instead of rshell.

When using rexec or rshell, it is important to keep in mind that although the command is executed with the privileges of the specified user, that user is not actually logged in. The user's login script is not executed and the program will not inherit the user's normal environment as it would during an interactive session. If you are connecting to a UNIX system, you should not attempt to execute programs which try to put standard input into raw mode; an example of this would be the vi editor. If you are connecting to a Windows system, you should not execute a program which uses a graphical interface. Only programs which read standard input and write to standard output are suitable for use with rexec or rshell.

The first step that your application must take is to initialize the library and then establish a connection. The following functions are available for use by your application:

RshInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other Remote Command API functions.

RshExecute
Execute the specified command on the server. The rshell or rexec protocol is selected based on the port number that is specified. Output from the command will be returned to the client to be read. When the command terminates, the connection to the server will be closed.

RshLogin

Establish an interactive login session which is similar to how the Telnet protocol works. If there is no host equivalence with the local host, you will be prompted for a password. Output from the session will be returned to the client, and when the client logs out the connection will be closed.

RshRead

Read the output generated by the command. Your application would typically call this function in a loop until all of the data has been read or an error occurs.

RshSearch

Search for a specific sequence of characters in the output returned by the server. The function returns when the sequence is encountered or when a timeout occurs. The data captured up to the point where the character sequence was matched is returned to the caller for processing.

RshDisconnect

Disconnect from the server and release the memory allocated for that client session. After this function is called, the client handle is no longer valid.

RshUninitialize

Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Secure Shell Protocol

The Secure Shell (SSH) protocol enables an application to establish a secure, interactive terminal session with a server, or execute commands remotely on the server, with the output of the command returned to the client. The SocketTools library supports both version 1.0 and 2.0 of the protocol.

The first step that your application must take is to initialize the library, then establish a connection to the server and authenticate the client. The following functions are available for use by your application:

SshInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call the application makes before calling the other SSH API functions.

SshConnect
Establish a connection to the server. This function will return a handle to a client session which is used in subsequent calls to the SSH API.

SshDisconnect
Disconnect from the server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

SshUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Connection Options

The SSH protocol has a number of advanced options which can be specified when establishing a connection. To provide additional information to the connection function, the SSHOPTIONDATA structure is used. The three most commonly used options specify if the connection will be used for an interactive terminal session or to execute a command on the server, and if a proxy server should be used when connecting to the server:

SSH_OPTION_TERMINAL
This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.

SSH_OPTION_COMMAND
This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudo-terminal is created for the client. The ***szCommandLine*** member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.

SSH_OPTION_PROXYSERVER
This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-related members of the SSHOPTIONDATA structure should be set to the appropriate values.

Input and Output

Once your application has connected to the server, any output generated by a program on the server will be sent as data for you to read. Any input to the program is sent by your application and received and processed by the server. The following functions are used:

SshPeek
Reads any data that has been sent by the server and copies it to the buffer provided by the caller, but

it does not remove the data from the internal receive buffer. This function can be used to examine the contents of the receive buffer and make decisions about how to process the data.

SshRead
Reads any data that has been sent by the server and copies it to a byte array buffer provided by your application. If the server closes the connection, the function will return a value of zero and the client can disconnect from the server at that point.

SshReadLine
Reads a line of text from the server, and returns it as a null terminated string. This function is useful for reading output from the server one line at a time. The function recognizes both UNIX and Windows end-of-line conventions, and will cause the application to block until a complete line of text has been read.

SshWrite
Send data to the server which will be received as input to the program. Your application provides the function with a byte array buffer that contains the data, and an integer value that specifies the number of bytes in the buffer.

SshWriteLine
Send data to the server as a line of text, terminated with a carriage-return and newline character. Note that your application should not specify the end-of-line characters, they are automatically sent to the server. This function will cause the application to block until the complete line of text has been written.

Command Processing

The SSH protocol can be used to connect to a server, log in and execute one or more commands, process the output from those commands and display it to an end-user using a graphical interface. The user never needs to see or interact with the actual terminal session. The SSH API provides functions which can simplify this kind of application, reducing the amount of code needed to process the data stream returned by the server.

SshExecute
This function executes a command on a server and copies the output to a user-specified buffer, with the exit code for the remote program as the function's return value. This is a convenience function that enables you to execute a remote command in a single call, without having to write the code to establish the connection and read the output.

SshGetExitCode
This function returns the exit code for the program that was executing on the server. It should be called after all of the data has been read and the server has closed the connection, which is indicated by the **SshRead** function returning a value of zero.

SshSearch
This function is used to search for a specific character or sequence of characters in the data stream returned by the server. The library will accumulate all of the data received up to the point where the character sequence is encountered. This can be used to capture all of the output from a command, or search for specific results returned by the command as it executes on the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The library provides an API for addressing and delivering messages, and extended features such as user authentication and delivery status notification. This library is typically used in conjunction with the Mail Message library to create the messages, and the Domain Name Service library to determine what servers are responsible for accepting mail for a specific user.

Mail Exchanges

When a message is delivered to a user, the application must determine what mail server is responsible for accepting messages for that user. This can be accomplished using the Domain Name Services (DNS) protocol, a protocol that is most commonly used to resolve host names such as www.microsoft.com into Internet addresses. This is typically accomplished by sending a request to a nameserver, a computer system that provides domain name services. In addition to resolving host names, nameservers can also provide information about those servers which are responsible for accepting mail for a given domain. There can be multiple servers which process mail for a domain with each server assigned a priority as part of their mail exchange (MX) record. If there is no mail exchange record for a domain, then the domain name itself is used.

To deliver a message directly to the recipient, you must examine the recipient address and request the list of mail exchanges for that user's domain. Using the DNS API, this is done by calling the **DnsEnumMailExchanges** function. If the recipient address is joe@example.com, you would want to enumerate the mail exchanges for the example.com domain. This will give you the name of the servers that will accept mail for users in that domain. For example, the function may return the host name mail.example.com as the name of the server which will accept mail for users in the example.com domain. Note that it is possible that one or more of the mail exchanges for a domain may not be in the recipient domain itself. In other words, it is possible that smtp.othercorp.net could be returned as a mail exchange for example.com. This is frequently the case when another organization is forwarding mail for that domain.

Therefore, there are three general steps that you must take when delivering mail directly to the recipient:

1. Parse the address of each recipient in the message. If you are using the MIME API, the **MimeEnumMessageRecipients** function can be helpful in extracting all of the recipient addresses. Everything after the atsign (@) in the address is the domain portion of that address.

2. Perform an MX record lookup using the DNS API function **DnsEnumMailExchanges** and specifying the recipient's domain. The function will return the name of the servers responsible for accepting mail for that user. If there are more than one server, they will be returned in order of their relative priority, with the highest priority server being returned first. This means that you should attempt to connect to those servers in the order that they are returned by the function.

3. Attempt to connect to the first server returned by the **DnsEnumMailExchanges** function. The connection should be on the default port, and you should not attempt to use any authentication. If the server accepts the connection, then use the **SmtpSendMessage** function to deliver the message. If the connection is rejected or the message is not accepted, attempt to connect to the next mail exchange server until all servers have been tried.

4. If no mail exchange servers were returned by **DnsEnumMailExchanges**, or you could not connect to any of them, attempt to connect to the domain specified in the address using the default port. If the connection succeeds, then deliver the message. If you cannot connect or the message is not accepted, then report to the user that the message could not be delivered.

One last important consideration is that many Internet Service Providers now block outbound connections on port 25 to any mail servers other than their own. If you are unable to establish any connections, either with the

error that the connection was refused or it consistently times out, contact your ISP to determine if port 25 is being blocked as an anti-spam measure. If this is the case, it will be required that you relay all messages through their mail servers or use an alternate port number.

Relay Servers

In some situations it may not be possible to send mail directly to the server that accepts mail for a given domain. The two most common situations are corporate networks which have centralized servers that are responsible for delivering and forwarding messages, or an Internet Service Provider (ISP) which specifically blocks access to all mail servers other than their own. This is usually done as either a security measure or as a means to inhibit users from sending unsolicited commercial email messages. If the standard SMTP port is being blocked, then any connection attempts will either fail immediately with an error that the server is unreachable, or the connections will simply time-out. In either case, a relay server must be specified in order to send email messages.

A relay server is a system which will accept messages addressed to users who may be in a different domain, and will relay those messages to the appropriate server that does accept mail for the domain. Using a relay server is generally easier than sending messages directly to the recipient. In order to send a message through a relay, you need to perform the following steps:

1. Connect to the relay server as you would normally.
2. Authenticate the client to the server. This may or may not be required, depending on how the server is configured. Some servers may be configured to only require authentication if you are connecting from an IP address that is not recognized as part of that system's network, for example, if you are connecting using a different Internet Service Provider. Others may always require authentication. Check with the server administrator if necessary to determine if and when authentication is required.
3. Use the **SmtpSendMessage** function to deliver the message to the recipients through the relay server. If there are multiple recipients, you can use the MIME API to enumerate the recipient addresses and then pass them to the **SmtpSendMessage** function.

It is important to note that using a mail server as a relay without the permission of the organization or individual who owns that server may violate Acceptable Use Policies and/or Terms of Service agreements with your service provider. Systems which relay messages from anyone, regardless of whether the message is coming from a recognized domain, are called open relays. Because open relays are often used to send unsolicited email, many administrators block mail that comes from one. It is recommended that users check with their network administrators or Internet service providers to determine if access to external mail servers is restricted and what is the acceptable use policy for relaying messages through their mail servers.

The first step your application must take is to initialize the library, then establish a connection to the server and authenticate the client if necessary. The following functions are available for use by your application:

SmtpInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other SMTP API functions.

SmtpConnect
Establish a connection to the SMTP server. This function will return a handle to a client session which is used in subsequent calls to the SMTP API.

SmtpAuthenticate
Authenticate the client session to the server using a username and password. This function should be called immediately after the connection has been established to the server. This is typically required if you are attempting to use the mail server as a relay, asking it to forward the message on to the server that actually accepts email for the recipient. Many Internet Service Providers (ISPs) require that users

authenticate prior to sending mail through their servers. You may need to contact the server administrator to determine if authentication is required.

SmtpDisconnect
Disconnect from the SMTP server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

SmtpUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Message Delivery

There are two general methods that can be used to deliver messages through the mail server. In most cases, it can be done with a single function call. However, there are some circumstances where it would be more appropriate to perform the transaction in stages. The SMTP API supports both methods.

SmtpSendMessage
This is the simplest method for sending an email message through the server. You provide the sender and recipient addresses, along with the message contents and the function will submit the message to the server for delivery.

SmtpCreateMessage
This function begins a transaction in which a message is dynamically composed, addressed and delivered in stages. You provide the sender address and message size to this function, and after it returns you begin the next stage, which is addressing the message.

SmtpAddRecipient
This function adds a recipient address to the recipient list for the message. This should be called once for each recipient, as well as for any recipients who are to receive "blind copies" of the message. A blind copy is when the message is sent to a recipient, but that recipient's address is not listed in any of the headers of the message; the other recipients will be unaware that the message was delivered to him. Most servers have a limit of approximately 100 recipients per message. It is possible that this function will return an error for a specific recipient address; the address may be malformed or it may not be acceptable for some other reason. This does not mean that the message will be rejected in its entirety, only that the specified recipient is not acceptable.

SmtpAppendMessage
This function should be called after all of the recipients have been added. It is used to send the contents of the message to the server. It is also possible to use the lower level **SmtpWrite** function to send data directly to the server, however **SmtpAppendMessage** is generally easier to use and can write data from memory, the system clipboard or from a file on disk.

SmtpCloseMessage
This function is called after the entire message has been sent to the server. This terminates the transaction and the message is submitted for delivery. Note that it is possible for the server to accept the message up to this point and then reject it at this final step due to some restriction, such as the message being too large.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench

The SocketWrench library provides an interface to the Windows Sockets API. It was designed to be simpler to use, and to provide functions which eliminate much of the redundant coding common to Windows Sockets programming. Developers who are working in languages other than C or C++ will find SocketWrench to be particularly useful because it does not use many of the complex structures that the Windows Sockets API uses. SocketWrench also supports creating client and server applications which use TLS security protocol without any dependencies on third-party security libraries.

The first step your application must take is to initialize the library. After the library has been initialized, the application can either take on the role of a client and establish a connection to a server, or become a server and listen for incoming connections from other clients.

InetInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call the application makes before calling the other SocketWrench API functions.

InetConnect
Establish a connection with a server. This function will return a handle to the application which can be used to send and receive data. When an application calls this function, it will be acting as a client. For an asynchronous session, use the **InetAsyncConnect** function.

InetListen
Begin listening for incoming client connections. This function will return a handle which should be passed to the **InetAccept** function to accept any clients which establish a connection. When an application calls this function, it will be acting as a server. For an asynchronous session, use the **InetAsyncListen** function.

InetAccept
Accept a connection from a client. This function should only be called if the application has previously called **InetListen**. If there is no client waiting to connect at the time this function is called, it will block until a client connects or the timeout period is reached. For an asynchronous session, use the **InetAsyncAccept** function.

InetUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Input and Output
When a TCP connection is established, data is sent and received as a stream of bytes. The following functions can be used to send and receive data over the socket:

InetRead
Read data from the socket and copy it to the memory buffer provided by the caller. If the server closes the connection, this function will return zero after all the data has been read. If the function is successful, it will return the actual number of bytes written.

InetIsReadable
This function is used to determine if there is data available to be read from the socket.

InetWrite
Write data to the socket. If the function succeeds, the return value is the number of bytes actually written.

InetIsWritable
This function is used to determine if data can be written to the socket. In most cases this will return a

non-zero value (True), unless the internal socket buffers are full.

Server Interface

The library provides a collection of functions which can be used to easily create a scalable, event-driven multithreaded server application. The server runs on a separate thread in the background, automatically managing the individual client sessions as servers connect and disconnect from the server. The application is notified of events through a callback mechanism, where it can respond to notifications such as a client establishing a connection with the server, or the client sending data to the server. Functions such as **InetRead** and **InetWrite** are used to exchange data with the clients. Because each client session is managed in its own thread, applications can perform calculations, access database resources and so on without worrying about interfering with other client sessions or the application's main UI thread.

InetServerStart

This function starts the server, creating the background thread and listening for incoming client connections on the specified port number. A socket handle is returned to the caller which is used to query the status of the server and perform other functions such as suspending and restarting the server.

InetEnumServerClients

This function enables the server application to determine the number of active client sessions, and obtain socket handles for each client that is connected to the server.

InetServerBroadcast

Broadcasts data to each of the clients that are connected to the server. This can be useful when the application needs to send the same data to each active client session, such as broadcasting a shutdown message when the server is about to be terminated.

InetServerThrottle

This function is used to control the maximum number of clients that may connect to the server, the maximum number of clients that can connect from a single IP address and the rate at which the server will accept client connections. By default, there are no limits on the number of active client sessions and connections are accepted immediately. This function can be useful in preventing denial-of-service attacks where the attacker attempts to flood the server with connection attempts.

InetServerSuspend

This function instructs the server to temporarily suspend accepting new client connections. Existing connections are unaffected, and any incoming client connections are queued until the server is resumed. It is not recommended that you leave a server in a suspended state for an extended period of time. Once the connection backlog queue has filled, any subsequent client connections will be automatically rejected.

InetServerResume

This function instructs the server to resume accepting client connections after it was suspended. Any pending client connections are accepted after the server has resumed normal operation.

InetServerLock

This function locks the server so that only the current thread may interact with the server and the client sessions. This will cause all other client threads to go to sleep, waiting for the server to be unlocked. This should only be used when the server application needs to ensure that no other client threads are performing a network operation. For general purpose synchronization, it is recommended that the application create a critical section rather than lock the entire server. If the server is left in a locked state for an extended period of time, it will cause the server to become non-responsive. If the application has started multiple servers, only one server can be locked at any one time.

InetServerUnlock

This function unlocks a server that has been previously locked. The threads which manage the client sessions will awaken and resume normal execution.

InetServerRestart
This function will terminate all active client connections, close the listening socket and re-create a new listening socket bound to the same address and port number. The function will return a socket handle for the new listening socket that should replace the original socket that was allocated using the **InetServerStart** function.

InetServerStop
This function will terminate all active client connections, close the listening socket and terminate the background thread that manages the server. Any incoming client connections will be refused, and all resources allocated for the server will be released.

Address Conversion

Internet Protocol (IP) addresses can be represented in one of two ways, either as unsigned 32-bit integer value or as string where each byte of the address is written as an integer value and separated by periods. For example, the local loopback IP address can either be specified as the string "127.0.0.1" or as the integer value 16777343. In most cases, using the string form of the address is easier; however, some functions require that the numeric value be used. The following functions are provided to enable you to convert between the two formats.

InetGetAddress
Convert an IP address string in dotted notation into a 32-bit integer value.

InetFormatAddress
Convert a numeric IP address into a string in dotted notation, copying the result into a buffer that you provide to the function.

Host Tables

When resolving a host name or IP address, the library will first search the local system's host table, a file that is used to map host names to addresses. On Windows 95/98 and Windows Me, if the file exists it is usually found in C:\Windows\hosts. On Windows NT and later versions, it is found in C:\Windows\system32\drivers\etc\hosts. Note that the file does not have an extension.

InetGetDefaultHostFile
Return the full path of the file that contains the default host table for the local system. This can be useful if you wish to temporarily switch between the default host file and another host file specific to your application.

InetGetHostFile
Return the full path of the host table that is currently being used by the library. Initially this is the same as the default host table for the local system.

InetSetHostFile
Specify a new host table which the library should use to resolve host names and IP addresses. This can be used by an application to provide its own local cache of host names and addresses in order to speed up the process of host name resolution.

Host Name Resolution

The library can be used to resolve host names into IP addresses, as well as perform reverse DNS lookups converting IP addresses into the host names that are assigned to them. The library will search the local system's host table first, and then perform a nameserver query if required.

InetGetHostAddress
Resolve a host name into an IP address, returned as a string in dotted notation. The library first checks

the system's local host table, and if the name is not found there, it will perform a nameserver query for the A (address) record for that host.

InetGetHostName
Resolve an IP address into a host name. The address is passed as a string in dotted notation, and the fully qualified host name is returned in a string buffer you provide to the function. The library first checks the system's local host table, and if the address is not found there, it will perform a nameserver query for the PTR (pointer) record for that address.

Local Host Information

Several functions are provided to return information about the local host, including its fully qualified domain name, local IP address and the physical MAC address of the primary network adapter.

InetGetLocalName
Return the fully qualified domain name of the local host, if it has been configured. If the system has not been configured with a domain name, then the machine name is returned instead.

InetGetLocalAddress
Return the IP address of the local host. If a valid socket handle is provided, then the IP address of the network adapter that was used to establish the connection will be returned. This can be particularly useful for multihomed systems that have more than one adapter and the application needs to know which adapter is being used for the connection.

InetGetAdapterAddress
Return the IP or MAC address assigned to a network adapter on the local system.

InetEnumNetworkAddresses
Enumerate the network addresses that are configured for the local host. If the system is multihomed, then the IP address for each network adapter will be returned.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol

The Telnet Protocol library enables an application to connect to a Telnet server, which provides an interactive terminal session similar to how character based consoles and terminals work. The user can login, enter commands and interact with applications programmatically or in conjunction with the terminal emulation library.

The first step that your application must take is to initialize the library, then establish a connection to the server and authenticate the client. The following functions are available for use by your application:

TelnetInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call the application makes before calling the other Telnet API functions.

TelnetConnect
Establish a connection to the Telnet server. This function will return a handle to a client session which is used in subsequent calls to the Telnet API.

TelnetDisconnect
Disconnect from the Telnet server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

TelnetUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call that the application should make prior to terminating.

Input and Output

Once connected to the Telnet server, any output generated by a program on the server will be sent as data for the client to read. Any input to the program is sent by the client and received and processed by the server. The following functions are used:

TelnetRead
Reads any output that has been generated by the program executing on the server. When the client first connects, the server typically executes a login program that requests the users authenticate themselves by entering a user name and password. Once the user has logged in, they are usually given a command line prompt where they can enter commands to be executed on the server. If the server closes the connection, the **TelnetRead** function will indicate that with an error result and the client can disconnect from the server at that point.

TelnetWrite
Send data to the Telnet server which will be received as input to the program. If the local echo option is enabled, then the client is also responsible for writing the input data to the display device, if there is one. If local echo is not enabled, the server will automatically echo back any characters written as data to be read by the client.

Telnet Modes

Telnet supports several modes of operation and the option negotiation phase, which occurs when a connection is established, is handled automatically by the library. There are two key modes which affect how the client session works:

TELNET_MODE_LOCALECHO
If this mode is enabled, it is the responsibility of the client to echo any data that it is sending to the server. For example, if the character "A" is sent to the server, the application must also send the character "A" to whatever interface the user is interacting with, such as a terminal emulation window. The default mode is for this option to be disabled, which means that the server will echo back any

data that is sent to it.

TELNET_MODE_BINARY
If this mode is enabled, the data between the client and server is not buffered and the high bit is not removed from any characters. If the application is executing a program which uses text mode windowing features (i.e.: it draws boxes on the display) then this mode must be enabled to ensure that the client processes the data correctly and it isn't buffered a line at a time. If this mode is disabled, then the data exchanged between the client and server will be buffered a line at a time and any 8bit characters will be stripped. This mode is enabled by default.

Command Processing

The Telnet protocol can be used to connect to a server, log in and execute one or more commands, process the output from those commands and display it to an end-user using a graphical interface. The user never sees or interacts with the actual terminal session. The Telnet API provides functions which can simplify this kind of application, reducing the amount of code needed to process the data stream returned by the server.

TelnetLogin
This function is used to automatically log a user in, using the specific user name and password. This function is specifically designed for UNIX based servers or Windows servers which emulate the same basic login sequence.

TelnetSearch
This function is used to search for a specific character or sequence of characters in the data stream returned by the server. The library will accumulate all of the data received up to the point where the character sequence is encountered. This can be used to capture all of the output from a command, or search for specific results returned by the command as it executes on the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation

The Terminal Emulation library provides a virtual terminal interface for emulating an ANSI or DEC VT-220 compatible character-based terminal. It can be used in conjunction with the Telnet API or the Remote Command API to display the output of commands executed on a server. It can also be used independently of any other networking library, such as providing emulation services for a serial connection.

The first step your application must take is to initialize the library, and then create a virtual display which is attached to a window. The following functions are available for use by your application:

NvtInitialize
Initialize the library for the current process. This must be the first function call the application makes before calling the other emulator API functions.

NvtCreateDisplay
Creates a new virtual display and returns a handle which can be used to reference that display. When creating the display, you provide a handle to a window the display will draw on, a handle to a font that will be used to display text characters, the type of emulation and the size of the display.

NvtDestroyDisplay
Destroy the virtual display, releasing the memory allocated for the handle. After this function returns, the display handle is no longer valid.

NvtUninitialize
Release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Display Management

The library provides functions to manage and update the virtual display, including a number of lower level functions that provide direct access to the display buffer. The most commonly used functions are:

NvtWriteDisplay
This is the most commonly used method of writing to the display. This function will automatically parse the data being written for escape sequences and update the display appropriately.

NvtUpdateDisplay
This function updates the window attached to the virtual display. This function should be called whenever a WM_PAINT message is received for that window, indicating that it needs to be redrawn. The window pain message can be handled in one of two general ways. The simplest is to just call **NvtUpdateDisplay** and return. The function will automatically acquire a device context for the window, redraw the portions of the window that need to be updated, and then release the device context. If you use this method, you should not call **BeginPaint** or **EndPaint**, since that is being done for you. If you wish to acquire the device context yourself by calling **BeginPaint**, then you must call **NvtSetDisplayDC** to attach the device context to the virtual display. When **NvtUpdateDisplay** is called, it will use that device context rather than acquiring one itself. After the display has been updated and you've called **EndPaint**, you should call **NvtSetDisplayDC** again, passing the function a null handle to detach the display from the device context that you've created.

NvtRefreshDisplay
Refresh the virtual display, updating the current cursor position and caret. The library will periodically refresh the display automatically based on its own internal state, but the application can call this if it wishes to force the display to refresh at that time.

NvtSetDisplayEmulation
This function can be used to change the emulation used by the library. Supported emulation types are

standard ANSI, DEC VT-100 and VT-200. It is also possible to specify that no emulation should be used, in which case the application is responsible for handling any control or escape sequences in the data being written to the display.

NvtResetDisplay
This function can be used to change the handle of the window associated with the display, the font being used and the size of the display. Resetting the display causes the contents of the display to be cleared.

NvtSetDisplayFont
This function sets the font which is used by the library to draw text on the display window. If you wish to specify a font name and point size instead of creating a font handle, use the **NvtSetDisplayFontName** function instead.

NvtSetDisplayMode
The emulation library has a number of modes which determines how the data is displayed, as well as controlling aspects of the window itself. This function can be used to enable or disable those modes as necessary. For the list of available modes, please consult the Technical Reference.

NvtSetDisplayColor
This function controls what colors are used when drawing the background of the display window as well as the default color of the text. In ANSI and VT-220 emulation, the color of individual characters can be specified by using escape sequences. Those colors can be modified by changing the display color map.

NvtSetDisplayColorMap
This function changes the default colors which are used when escape sequences are used to change the foreground or background color of a character cell. In most cases the default color map will be appropriate, but applications can change the RGB values associated with an entry in the color map if needed. For example, the default value for the color gray is at position 8 in the color map index with an RGB value of 192,192,192. If you wanted to use a darker color, you could change the RGB value 128,128,128.

Cursor Control

There are a number of lower level functions which enable an application to have direct control over cursor positioning, clearing the display and so on. In most cases these functions are called automatically by the library as the result of processing the control and escape sequences found in the data being written to the display. However, an application can call these functions directly in order to manage the display itself. One important thing to keep in mind is that the X,Y positions used by these functions refer to the cursor position in the virtual display and correspond to columns and rows, not pixels.

There is also a slight difference in terminology that you should be aware of when reading the technical reference documentation. In Windows, the term "cursor" is typically used to refer to the mouse pointer, while "caret" is used to refer to the blinking marker that is displayed at the current position in the display. In the documentation for the emulator API, the term "cursor" is used in the same way that it is used for character based terminals, as the marker for the current position in the display. Therefore, in terms of the emulation API, you can think of the cursor and the caret as being synonymous.

NvtGetCursorPos
Return the current position of the cursor in the virtual display.

NvtSetCursorPos
Change the current position of the cursor in the virtual display. This function will normalize the X,Y values to be bound by the size of the display. If a scrolling region has been set, the coordinates will be bound by that region. The upper left corner of the display is 0,0.

NvtSaveCursor
Saves the current position of the cursor. The cursor position can be restored by calling the **NvtRestoreCursor** function.

NvtClearDisplay
Clear the contents of the display. You can clear from the start of the display to the current cursor position, from the current position to the end of the display or the entire display.

NvtDeleteChar
This function deletes a character from the current cursor position, shifting the remaining characters on the line to the left.

NvtDeleteLine
This function deletes the line at the current cursor position, shifting the remaining lines in the display up.

NvtEraseChar
This function erases a character at the current cursor position without affecting the characters that follow it on the line.

NvtEraseLine
This function erases the line at the current cursor position without affecting the lines that follow it.

NvtInsertChar
Insert a character at the current cursor position, shifting the following characters to the right.

NvtInsertLine
Insert a blank line at the current cursor position, shifting the following lines down.

NvtGetDisplayAttributes
Return the current attributes which have been enabled. Possible attributes are reverse, bold, dim, underline, hidden and protected. These correspond to the attributes that are commonly used with character based terminals.

NvtSetDisplayAttributes
Set the default attributes which are used when characters are written to the display. For example, setting the attribute NVT_ATTRIBUTE_REVERSE would cause all subsequent characters to be displayed with the foreground and background colors reversed. This would continue until the display attributes were set back to NVT_ATTRIBUTE_NORMAL.

Function Key Mapping

Another aspect of terminal emulation is how function keys and other special keys are handled by the application. The emulation library provides functions which will convert Windows virtual key codes into the escape sequences that are generated by character based terminals.

NvtTranslateMappedKey
This function translates a virtual key code into the escape sequence which should be sent to the server to emulate pressing that key. For example, if the user presses the F1 key on the keyboard, this will generate a WM_KEYDOWN event with VK_F1 as the virtual key code. This function will translate that key code into the three characters escape sequence ESC O P (the ASCII codes 27, 79, 80). That sequence of characters should be sent to the server, which will recognize it as the F1 function key being pressed. It is important to note that the different emulation types have different key mappings. Therefore, the server must be set to recognize the same type of terminal that you are emulating. If you have the emulation set as VT-220 but the server thinks that you are emulating a VT-100, it will not recognize some of the escape sequences correctly.

NvtGetMappedKey

Returns the escape sequence associated with a specific function key or special key. There are a total of 50 special keys recognized by the emulator. They consist of the F1 through F12 function keys, the function keys when they are in a shift state, the arrow keys and the keypad keys. Refer to the technical reference for the complete list.

NvtSetMappedKey
Change the escape sequence associated with a specific function key or special key. Changing this value will cause **NvtTranslateMappedKey** to return the new escape sequence when that key is pressed. This function can also be used to restore the default mapping for a key.

Windows Messages

Because the emulation library is closely tied to a display window, it is important to understand how Windows messages should be handled. If you are not familiar with how Windows processes messages at the API level, it is recommended that you review the technical reference material available from Microsoft on user interface programming. Another excellent resource is "Programming Windows" by Charles Petzold. If you are a C++ programmer, the equivalent MFC virtual functions will also be listed here.

WM_CREATE
This message is sent when the window is being created, but before it is displayed. This is typically a good point to call the **NvtCreateDisplay** function to create the virtual display and attach it to the window. You can also call **NvtSetDisplayMode** to set the various display modes, such as whether you want scrollbars to be shown, what kind of caret should be used and so on. In MFC, this can be done in the Create method for the window.

WM_DESTROY
This message is sent when a window is being destroyed. The application should call **NvtDestroyDisplay** at this point, since the window will no longer exist after the event handler returns.

WM_PAINT
This message is sent when the window needs to be redrawn. This is where the **NvtUpdateDisplay** function should be called. If you are using MFC, the code should be placed in the OnPaint method.

WM_HSCROLL and **WM_VSCROLL**
This message is sent when the user interacts with the window's scrollbars. An application should call the **NvtSetDisplayScrollPos** function to update the scroll position in the virtual display. In MFC, this code should be implemented in the **OnHScroll** and **OnVScroll** methods.

WM_KEYDOWN
This message is sent when the user presses a key. To determine if the key is mapped to a special escape sequence that should be sent to the server, call the **NvtTranslateMappedKey** function. In MFC, this code should be implemented in the **OnKeyDown** method.

WM_SETFOCUS and **WM_KILLFOCUS**
This message is sent when the window acquires or loses focus. You should call **NvtSetDisplayFocus** so that the library knows that the input focus has changed. In MFC, this code should be implemented in the **OnSetFocus** and **OnKillFocus** methods.

WM_SIZE
This message is sent when the window size changes. The application should call the **NvtResizeDisplay** function so that the virtual display is aware that the window size has changed. In MFC, this code should be implemented in the **OnSize** method.

When writing a terminal emulation program, it will also be necessary to handle other Windows messages such as WM_CHAR to process keys pressed by the user, which should be sent to the server. It is recommended that you refer to the Terminal sample program which demonstrates how these messages can be handled by an application. It also serves as a good example of how the Terminal Emulation API can be used in conjunction

with the Telnet API.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Library

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The Text Message API provides functions that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

The first step your application must take is to initialize the library, which will load the required system libraries and initialize the internal data structures that are used. You must call the initialization function before attempting to call any other function in the library.

SmsInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call that the application makes before calling the other SMS API functions.

SmsUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Text Messages

Sending a text message is done with a single function call, with two parameters. The first parameter is a pointer to a data structure that identifies the service that will be used to send the message. By default, messages are sent via an SMTP gateway, however the API was designed to be extensible so that additional methods could be integrated into future versions of the library. For example, a third-party company may offer a service that allows messages to be sent using HTTP and that can be added as an additional service type. The second parameter is a pointer to a data structure that contains information about the text message itself.

SmsSendMessage
This function is used to send the text message. The caller must populate two data structures that contain information about the service provider and the contents of the message itself. The SMSSERVICE structure is used to provide information about the service being used to send the message, and the SMSMESSAGE structure is used to provide the function with information about the message.

Service Providers

When a service provider is mentioned in the documentation, typically it is referring to the wireless service provider (also commonly called a "carrier") that is responsible for providing network access for the mobile device. These are identified by name, such as "Verizon Wireless" and "AT&T Mobility". The library has a built-in table of known providers in North America, and can return this information to your application. Note that in some cases, a service provider may also refer to a specific service used to send a text message.

SmsGetProvider
This function returns information about the wireless service provider that is associated with a specific phone number. This can be used to determine if a phone number is valid, and the default provider that is responsible for that number. Information about the service provider is returned in an SMSPROVIDER structure.

SmsEnumProviders
This function enumerates all of the supported wireless service providers and populates an array of structures that contain information about each provider. Typically this is used to update the user interface with a list of known service providers, allowing the end-user to select a specific service provider.

SmsGetFirstProvider and SmsGetNextProvider

These functions are also used to enumerate the supported wireless service providers, but they return information about a single provider with each function call. They are primarily designed to be used with languages that don't provide an easy way to work with an array of data structures. The information that is returned is identical to the **SmsEnumProviders** function.

Gateway Servers

A gateway server refers to the server that is responsible for accepting the text message and sending it to the recipient. Currently, this is exclusively used in the context of SMTP gateways where the message is sent to a mail server operated by the wireless service provider.

SmsGetGateway

This function returns information about the gateway that is responsible for accepting a text message for a specific phone number and forwarding that message to the mobile device. Information about the gateway is returned in an SMSGATEWAY structure.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Web Storage

The Web Storage library provides a private cloud storage API for uploading and downloading shared data files which are available to your application. This is primarily intended for use by developers to store configuration information and other data generated by the application. For example, you may want to store certain application settings, and the next time a user or organization installs your software, those settings can be downloaded and restored.

The connection to the storage service is always secure, using TLS 1.2 and AES-256 bit encryption. There are no third-party services you need to subscribe to, and there are no additional usernames or passwords for you to manage. Access to the service is associated with an account which is created when you purchase a development license, and the security tokens are bound to the runtime license key used when initializing the API. You also have the option to compress and encrypt your data you store using the Encoding, Compression and Encryption APIs.

Terminology

When you get started with the storage API, you'll notice there is some different terminology which is used. This will provide an overview of that terminology, and compare it to common terms used with traditional protocols like FTP. When accessing an FTP server, you generally deal with ***directories***, ***files***, ***names*** and ***types*** (generally whether the file is binary or text). The storage API has similar concepts, but uses somewhat different terminology.

Application Identifiers

An application identifier (AppId) is a null-terminated string which uniquely identifies your application. This string, used in conjunction with your runtime license key, is used to generate an access token. This token is used to access the storage container which contains the data which you've stored.

It is recommended you use a standard format for the AppId which consists of your company name, application name and optionally a version number. Some examples of an AppId string would be:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

It is important to note with these two example IDs, although they are similar, they reference two different applications. Objects stored using the first ID will not be accessible using the second ID. If you want to store objects which should be shared between all versions of the application, it is recommended you use the first form, without the version number. If you want to store objects which should only be accessible to a specific version of your application, then it is recommended you use the second form which includes the version number.

The AppId must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of the string is 64 characters, including the terminating null character. It is not required for your application to create a unique AppId. Each storage account has a default internal AppId named `SocketTools.Storage.Default`. This AppId is used if a NULL pointer or an empty string is specified.

Containers

Storage containers are somewhat analogous to directories or folders in a file system, however they are general purpose and designed to allow you to control how your application accesses the data that's been stored. There are four container types which are defined by the API, and you can think of them as types of boxes or file cabinets which you store your data in.

It is important to keep in mind these containers are available to all users of your application, your program controls who has access to any particular data file. Your users will not be able to "browse" any of the containers unless you specifically provide that capability by implementing it in your own code. There is no public access to any of the data which you upload, and our service does not use an open API accessible by third parties.

`WEB_STORAGE_GLOBAL`
The global storage container which is available to all users of your application. Any data stored in this container is available to everyone who uses your software. Unless you have a specific need to limit access to the data to a specific user or group of users, this is the recommended container you use to store data.

`WEB_STORAGE_DOMAIN`
The domain storage container is limited to users in the same local domain, defined either by the name of the domain or workgroup assigned to the computer system. This can provide a kind of organization wide storage, but it does depend on the domain being unique. For example, if you are using domain storage for your application, and you have multiple customers who have systems part of the default "Workgroup" domain, they would all share the same container. If the domain or workgroup name changes, then data stored in the container would no longer be available.

`WEB_STORAGE_MACHINE`
The local machine storage container is associated with the physical computer system your application is running on. The machine is identified by unique characteristics of the system, including the boot volume GUID. Data stored in this container can only be accessed from the application running on that particular system. If the operating system is reinstalled, the machine ID will change and data stored in this container would no longer be available.

`WEB_STORAGE_USER`
The current user storage container is associated with the current user who is using your application. The user identifier is based on the Windows Security Identifier (SID) assigned to the account when it's created. If the user account is deleted, the data stored in this container will no longer be available to the application. Another user on the same computer system would not be able to access the data in this container.

If you decide to use anything other than global storage, the data your application stores can be orphaned if the system configuration or user account changes. It's recommended you store critical application data and general configuration information using **`WEB_STORAGE_GLOBAL`** and use other non-global storage containers for configuration information which is unique to that system and/or user which is not critical and can be easily recreated. If you're concerned about protecting the data you upload to global storage, you can encrypt it prior to storing it.

Objects

Storage objects are similar to files in a file system. They are discrete blocks of data, associated with a label (name), have attributes and are associated with a particular content type. However, an object does not need to be an actual file on the local system. For example, you could store an object which is a string, a pointer to a structure, or any block of memory. You could also just store a complete file as an object. Unlike files, you cannot perform partial reads of an object or "seek" into certain parts of a stored object. Of course, you can download an object, either in memory or to a local file, and perform whatever operations you require on the data.

Labels

Object labels are similar to file names, and are a way to identify a stored object instead of using its internal object ID. However, there are some important differences. The most significant difference

being labels are case-sensitive, unlike Windows file names. An object with the label "AppConfig" is considered to be different than one with the label "appconfig". Labels can contain Unicode characters, but they cannot contain control characters.

You can also use forward slashes or backslash characters in the label, but it's important to note objects are not stored in a hierarchical structure. Your application can store objects using a folder-like structure, but it's not something which is enforced by the API.

Media Type

Each object your application stores is associated with a media type (also called a content type) which identifies the object's data. This uses the standard MIME media type designations, such as "text/plain" or "application/octet-stream". Your application can explicitly specify the media type you want to associate with the object, or you can have the API choose for you, based on the contents of the object and using the label as a hint for what it may contain. For example, if you create an object with the label "AppConfig.xml" and it contains text, then the API will select "text/xml" as the default media type.

Initialization

The first step your application must take is to initialize the library and then open a storage container. The following functions are available for use by your application:

WebInitialize

Initialize the library for the current process. This must be the first function call the application makes before calling the other service API functions.

WebOpenStorage

Opens a storage container for your application and returns a handle which is used with other functions.

WebCloseStorage

Close the storage container and release the resources allocated for the session.

WebUninitialize

Release all resources which have been allocated for the current process. This is the last function call the application should make prior to terminating.

Data Storage

The library provides functions to upload and download to the storage container. You can store local files, or you can create objects from data in memory. There are also functions which make it easier to store and retrieve textual data using null-terminated strings.

WebGetFile

Download the object data and store in a file on the local system. There is also an extended version of the function named WebGetFileEx which provides additional options, such as the ability to retrieve information about the object downloaded in a single function call.

WebGetObject

Download the object data and store it in a memory buffer provided by the caller. The buffer can either be a pre-allocated block of memory, or reference a global memory handle which will contain the data when the function returns.

WebGetTextObject

Download the object data and store it in a null-terminated string. This function would be used with data which is only textual, and it may contain Unicode text. If the Unicode version of the function is called, the text is automatically converted from UTF-8 encoded to UTF-16 encoded text.

WebPutFile

Upload the contents of a local file and store it as an object in the current container. There is also an extended version of the function named WebPutFileEx which provides additional options, such specifying the object attributes, media type and information about the stored object when the function returns.

WebPutObject
Upload the object data from a buffer in memory and store it as an object in the current container. This function would typically be used to store binary data, including compressed or encrypted text.

WebPutTextObject
Upload the contents of a string and store it as an object in the current container. The text is specified as a null-terminated string, and if the Unicode version of the function is used, the text will be converted and stored as UTF-8 encoded text. The WebGetTextObject function will automatically convert the text back to UTF-16 encoding when the text object is retrieved from storage.

Data Management

The data management functions allow you to obtain information about stored objects and perform typical operations such as copying, renaming and deleting objects from the container.

WebGetObjectInformation
Returns information about a specific object stored in the container. This function populates a WEB_STORAGE_OBJECT structure which will contains metadata for the object such as its ID, size, attributes, creation and modification times.

WebGetFirstObject
Enables your application to search for and enumerate objects in a container based on their label and/or their media type. This function is used in conjunction with the WebGetNextObject function to list all matching objects in a container.

WebCompareFile
Compares the contents of a local file with the data in a stored object. This function can be used to determine if the contents of a file have changed since the file was previously stored using the WebPutFile function.

WebCompareObject
Compares the contents of data in memory with the data in a stored object. This function can be used to determine if the contents of the buffer have changed since the data was previously stored using the WebPutObject function.

WebCompareText
Compares the contents of null-terminated string with the text in a stored object. This function can be used to determine if the contents of the string have changed since the data was previously stored using the WebPutTextObject function. If the Unicode version of this function is used, the string is automatically converted to use UTF-8 encoding prior to being compared with the contents of the stored object.

WebCopyObject
Copies the contents of a stored object to a new container, or duplicating the object within the same container using a different label. Information about the newly created object is returned to the caller.

WebMoveObject
Moves the contents of a stored object to a new container. Information about the object which has been moved will be returned to the caller.

WebRenameObject
Changes the label associated with a stored object. The new label for the object cannot already exist in

the same container. If you want to change the label to one already assigned to an existing object, the object must first be deleted.

WebDeleteObject
Removes the stored object from the container. This operation is immediate and permanent. Deleted objects cannot be recovered by the application at a later time.

WebGetStorageQuota
Returns information about how much storage space is available. It will populate a WEB_STORAGE_QUOTA structure which will specify how many bytes of storage you're using and how many objects you have created. It is important to note accounts created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

WebResetStorage
Resets the storage container and deletes all objects which were stored there. This function resets the container back to its initial state, deleting all object metadata from the database and removing all stored data. This operation is immediate and the objects stored in the container are permanently deleted. They cannot be recovered by your application.

Other Functions

Several additional utility functions are available as part of the storage API, including functions to register and de-register application identifiers and validate object labels.

WebRegisterAppId
Register a new application identifier (AppId) to be used to access a storage container. It is not required you create a unique application ID, but it can be helpful to distinguish stored content between different versions of your applications.

WebUnregisterAppId
Unregister an application identifier which was previously registered by your application. You should be extremely careful when using this function because it permanently delete all stored objects created using the AppId value. Internally it revokes the access token granted to your application and causes the server to expunge all objects in the container associated with the token.

WebValidateAppId
A utility function which can be used to validate an application identifier, ensuring it is valid and has been registered.

WebValidateLabel
A utility function which can be used to validate an object label to ensure it does not contain any invalid characters. This would be primarily used by applications which allow a user to specify the label names for the objects being stored.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebView Browser

The SocketTools WebView library embeds a modern Chromium-based browser into your application using Microsoft's WebView2 runtime. It provides a Win32-style API for hosting a browser in any window you manage, as well as creating a top-level browser window that the library owns. The focus is on simplicity without the need to interact with the underlying Edge WebView2 COM interface.

The API provides high-level functions for navigating to a URL and setting HTML content, as well as providing more granular control over options, headers, security state, downloads, zoom, and user experience. The API can be used with any language which supports calling functions in a standard Windows DLL and does not require installing the WebView2 SDK or including it in your project.

Initialization

To create an instance of the browser in your application, the library must be initialized and the WebView2 runtime must be available. These functions manage per-process startup and shutdown, as well as a quick availability check you can perform before initialization. After the library has been initialized, you're ready to either attach it to an existing host window you've created or request the library to create its own top-level application window to host the browser.

WebViewRuntimeAvailable
Check whether a compatible WebView2 runtime is present before initializing.

WebViewInitialize
Initialize the library for the current process. Call this once before using the WebView API.

WebViewUninitialize
Unload the library and release process-wide resources before your application terminates.

Browser Window

The WebView library provides two general approaches to creating an instance of a browser window. Your application can ask the WebView library to attach directly to a window handle that you provide. This can be any valid child window, such as a static control on a dialog or a custom view area within a frame window. After the browser is attached, the WebView content is rendered inside that region just like any other control. This approach is ideal for applications that already have a defined user interface and simply want to embed a modern HTML renderer inside an existing window.

Alternatively, your application can let the library create and manage its own top-level browser window. In this case, the library creates the window, handles its message loop, and automatically closes the WebView when the window is dismissed. This model is often used by utilities or tools that need to display web content without integrating it into an existing UI layout. It works equally well in graphical and console-based applications A console program can initialize the library, create a browser window, and then wait until the user closes it, without having to implement a GUI message loop of its own.

When a WebView is created or attached to a window, the rendering of the browser content is handled entirely by the WebView2 engine. The application does not need to implement any drawing logic or process paint messages for the control. All HTML, CSS, and script rendering occurs in a separate compositor process managed by the runtime, and updates are automatically synchronized with the window surface. From the application's perspective, the WebView behaves like a self-contained visual control and you simply position and show it like any other child window. This design eliminates the complexity of handling `WM_PAINT` or managing device contexts, ensuring smooth and efficient updates without requiring the host to perform any explicit drawing operations.

WebViewAttachWindow and WebViewAttachWindowEx
Create and attach a WebView instance to an existing window you own. The extended version of this function supports using an alternate language, specifying a user data folder, and other session

options. This should only be used with applications that already have a message pump that processes Windows messages.

WebViewCreateWindow
Create a top-level browser window managed by the library. This is useful when you want the WebView in its own window with a message loop. This can be used with any type of application, including console applications which need to create a browser.

WebViewDetachWindow / WebViewDestroyWindow
Cleanly detach from a host window or destroy a created browser window and its resources.

WebViewSetBounds
Set the size and position of the browser window within the host window. This is typically used in response to Windows messages like `WM_SIZE` when the user resizes your user interface.

WebViewSetBackgroundColor
Set the default background color, with support for transparency using options when the browser window is attached.

WebViewRedrawWindow
Forces the browser host and child window to redraw themselves. This clears the browser cache and causes the browser to reload the current document and render it again on the display. This can useful if the window needs to be completely redrawn due to changes in monitor DPI contexts.

WebViewShowWindow and WebViewIsVisible
Show or hide the view, query visibility, and set its position and size in device pixels.

WebViewGetZoomLevel and WebViewSetZoomLevel
Get or set zoom as a percentage for accessibility or kiosk scenarios.

WebViewSetOption
Enable or disable features like scripting, dialogs, context menus, messages, audio, accelerators, and more.

Navigation

The library supports standard HTTP and HTTPS URLs, as well as local files using the file:// scheme or direct Windows file paths. When you call a navigation function, the library automatically normalizes the specified URL, resolving relative paths, encoding unsafe characters, and converting local paths into properly formatted file:// URLs as needed. Optional flags allow you to perform additional safety checks during normalization, such as rejecting unsupported schemes or disallowing navigation to local files from remote content. This helps ensure that only trusted content is displayed, and reduces the risk of mixed-content or cross-origin issues.

Under the hood, navigation is handled by the Microsoft Edge (Chromium) engine, providing the same secure and standards-compliant networking stack used by the desktop browser. Connections automatically negotiate the most secure protocol supported by both client and server, with full support for HTTP/2 and HTTP/3 when available. The library also enforces TLS certificate validation and can report detailed security information for each page, including the certificate information and encryption strength. This ensures that applications embedding web content benefit from the same modern security features as a standalone browser.

WebViewNavigateUrl
Navigate to an Internet or local file URL and render the document.

WebViewSetDocumentText
Replaces the current document with the specified HTML or plain text.

WebViewGetDocumentText
Returns the contents of the current document being displayed in the browser.

WebViewGetFilteredText
Returns the current document as plain text with any HTML markup filtered out, while retaining links and other relevant information. If your application needs to check a page for specific text, using this function can simplify that process so you don't need to be concerned with markup and scripting content.

WebViewReloadDocument
Reload the current page, optionally clearting the browser cache.

WebViewCancelNavigation
Cancel the current navigation. This is typically done within an event handler to tell the browser it should stop loading the document.

WebViewGoBack and WebViewGoForward
Move through the navigation history in the same way that back and forward buttons work in a browser.

Event Handling

The library uses a single event-driven callback model to notify the application about browser activity. When you enable events for a WebView instance, all notifications, such as navigation start and completion, messages and downloads, are delivered through a single callback function. Each event provides an identifier, parameters, and optional context information that your application can use to respond appropriately. This design simplifies integration by eliminating the need for multiple COM-style event interfaces, while still giving you fine-grained control over what happens in response to browser actions. Events are processed on the same thread that created or attached to the browser.

WebViewEventProc
The callback function you implement to handle event notifications from the browser. Whenever an event occurs, such as a navigation starting or a download completion event, your callback function will be invoked. This is similar to the standard Win32 **WndProc** callback function where an event ID, a handle to the host window and additonal parameters are provided so you can take an action based on the type of event. You can provide the address of your callback function when the browser is created or attached to a window, and it can be set or changed later through the API.

WebViewEnableEvents and WebViewDisableEvents
Register a callback function for all WebView events or disable event handling for the session.

WebViewWaitForEvent and WebViewWaitUntilClosed
Wait for a specific event while processing messages or wait until a browser window is closed.

WebViewKeyEventHandled
Prevent a specific accelerator key event from being processed, allowing your application to take over that function.

Security

The library defaults to using secure HTTPS connections whenever possible, automatically enforcing modern TLS protocols and certificate validation. This ensures that data exchanged between the embedded browser and remote servers is encrypted and protected against tampering. When a page attempts to load both secure and non-secure content, commonly referred to as ***mixed content***, the browser may block or downgrade certain requests to maintain security. Developers should generally avoid mixing local or insecure resources with secure pages, as this can lead to unexpected warnings in the embedded browser. Your application can query detailed information about a page's security state,

including certificate details and encryption strength, to display status indicators or perform custom validation.

WebViewGetSecurityInfo
Retrieve detailed information about the page's current security state, such as the TLS protocol version, cipher suite, certificate subject and issuer, and the certificatevalidity period.

WebViewGetSecurityLevel
A higher level function which returns the current page's overall security state. A higher value indicates a higher level of security. This can be convenient for providing visual indicators to users, such as a padlock icon for secure sites. The security level is based on a combination of factors, such as if TLS was used to retrieve the content, if the certificate is valid and hasn't expired, and there is no mixed content.

Scripting and Messaging

Web pages displayed in a WebView can communicate directly with the host application using the standard **`window.chrome.webview.postMessage`** interface provided by the WebView2 runtime. When a page sends a message in this way, the library delivers it to the application through its registered event handler. The message can contain plain text or structured JSON data, allowing scripts to trigger application actions or report results back to native code, making it possible to create interactive hybrid interfaces. For example, a local HTML dashboard that requests data from your application or invokes native Windows functionality.

In the opposite direction, the host application can send messages to the page using the **WebViewPostMessage** function. These messages are received by the page's JavaScript code through an event listener attached to **`window.chrome.webview`**. This provides a simple, bidirectional channel for exchanging information between native code and the web environment. Applications can use it to push status updates, notify the page of user actions, or inject serialized data without reloading the document. Together, the scripting and messaging functions enable seamless integration between modern web content and traditional Windows applications.

WebViewExecuteScript and WebViewAsyncExecute
Execute JavaScript in the page and retrieve the result, either synchronously or asynchronously using an event callback.

WebViewGetResult
Get the result of a script which was executed in the context of the current browser page.

WebViewPostMessage
Post a JSON or string message from the host to the page. Your web app can listen on **`window.chrome.webview`** to receive messages.

WebViewGetMessage
Return the last message sent by the page to your native application using the **`window.chrome.webview.postMessage`** interface.

Saving and Printing

The library provides built-in support for saving and printing the current document using the browser's rendering engine. Applications can save the displayed content in several formats, including MHTML, HTML with supporting assets, PDF, or as an image capture of the current viewport. This makes it easy to archive pages, export reports, or capture visual output exactly as it appears on screen. When printing, the library uses the standard print interface provided by the browser, ensuring consistent output and access to all installed printers. The print dialog is managed by the WebView runtime rather than the application itself, providing users with the familiar print options they would expect from a modern web browser.

WebViewSaveDocument
Save the current page in a chosen format. This function supports saving the current document as text only, HTML text, MTHML or as a PDF. It also supports saving a snapshot of the current browser viewport as an image file in various formats such as JPEG and PNG.

WebViewPrintDocument
Show the system print dialog for the current document. This uses the same general printing interface as the Edge browser.

File Downloads

When a user begins downloading a file, the library generates events that allow the application to monitor progress, query download information, and specify the final save location. Because downloads are handled internally by the WebView2 runtime, the application does not need to implement its own network or file transfer logic. This makes it easy to integrate secure, managed downloads into your application while retaining information about each file's transfer state.

WebViewEnumDownloads
Enumerate the files which have been downloaded during the current browser session. Each file is identified using a unique integer ID and this can be passed to other functions to return detailed information about the current state of the transfer. You can enumerate all file transfers or specific downloads based on their state. For example, you can enumerate the files when a download is in progress, or only those files which have been successfully downloaded.

WebViewGetDownloadInfo
Returns information about a specific file transfer, based on the file transfer ID provided through an event handler or by enumerating the downloads.

WebViewSetDownloadPath
Change the path of the file being downloaded, specifying a new location to store the file on the local system. By default this is the current user's Downloads folder; however, your application can use this function to change the location to a differet folder.

WebViewCancelDownload
A download in progress can be canceled using this function. It can also be used to prevent a file from being downloaded at all by canceling the transfer in an event handler. For example, if the file name matches a certain pattern (such as a particular file extension), you can use this function to prevent the transfer from initating. In this case, the browser will not show its standard download dialog in the upper right of the window.

Browser Session

The library provides functions to manage the browser session, including setting custom request headers, clearing the cache and getting the value of headers set by the server in response to a navigation request.

WebViewGetFirstHeader and WebViewGetNextHeader
Enumerates all of the headers set by the server in response to a navigation request. This is used after the current navigation has completed.

WebViewGetHeader
Get the value of a specific header set by the server.

WebViewSetHeader
Set a custom request header and its value to be included with subsequent navigation requests.

WebViewGetUserAgent and WebViewSetUserAgent
Get and set the standard User-Agent header value that is sent with every request.

WebViewClearCache and WebViewClearCookies and WebViewClearHistory
Clear the cached data, cookies and navigation history for the current session.

WebViewSetTimeout
Adjust the amount of time the browser waits for a navigation request to complete.

Utility Functions

WebViewResetBrowser
Reset per-navigation state and cached details such as messages and script results when you need a clean slate.

WebViewNormalizeUrl
Normalize a URL entered by a user and optionally perform addition checks to ensure safety.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Whois Protocol

The Whois protocol library provides an interface for requesting information about an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The Whois protocol enables an application to query a server that provides that registration information. The Whois library provides an API for requesting that information and returning it to the program so it can be displayed or processed.

The first step your application must take is to initialize the library and then establish a connection. The following functions are available for use by your application:

WhoisInitialize
Initialize the library and load the Windows Sockets library for the current process. This must be the first function call the application makes before calling the other Whois API functions.

WhoisConnect
Connect to the server, using either a host name or IP address. This function returns a client handle which is used in subsequent calls to the library.

WhoisSearch
Perform a search for a specific domain. The server will return the results of the search as a text document which provides registration information for that domain. It is important to note that different registrars may use different formats when returning the data, and not all servers return the same type of information.

WhoisRead
Read the data returned by the server. Your application would typically call this function in a loop until all of the data has been read or an error occurs.

WhoisDisconnect
Disconnect from the server and release the memory allocated for that client session. After this function is called, the client handle is no longer valid.

WhoisUninitialize
Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last function call the application should make prior to terminating.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Class Library Reference

- Domain Name Service Class
- File Transfer Protocol Class
- File Transfer Queue Class
- File Transfer Server Class
- Hypertext Transfer Protocol Class
- Hypertext Transfer Queue Class
- Hypertext Transfer Server Class
- Internet Control Message Protocol Class
- Internet Message Access Protocol Class
- Internet Server Class
- Large Language Model Class
- Mail Message Class
- Message Store Class
- Network News Protocol Class
- Network Time Protocol Class
- News Feed Class
- Post Office Protocol Class
- Remote Command Class
- Simple Mail Transfer Protocol Class
- Secure Shell Protocol Class
- SocketWrench Class
- Telnet Protocol Class
- Terminal Emulation Class
- Text Message Class
- Web Storage Class
- WebView Browser Class
- Whois Protocol Class

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Class Library

Resolve domain names into Internet addresses and return information about a remote host, such as the servers that are responsible for accepting mail for the domain.

Reference

- Class Methods
- Constants
- Data Structures
- Error Codes

Library Information

Class Name	CDnsClient
File Name	CSDNSV12.DLL
Version	12.0.3024.1286
LibID	C887CFA2-F7A7-4A9F-8AF2-48ED79642023
Import Library	CSDNSV12.LIB
Dependencies	None
Standards	RFC 1034

Overview

The Domain Name Services (DNS) protocol is what applications use to resolve domain names into Internet addresses as well as provide other information about a domain. All of the SocketTools libraries provide basic domain name resolution functionality, but the Domain Name Services library gives an application direct control over what servers are queried, the amount of time spent waiting for a response and the type of information that is returned.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of

version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Class Methods

Class	Description
CDnsClient	Constructor which initializes the current instance of the class.
~CDnsClient	Destructor which releases resources allocated by the class.
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class.
AttachThread	Attach the specified client handle to another thread.
Cancel	Cancel an outstanding nameserver query.
DetachHandle	Detach the handle for the current instance of this class.
DisableTrace	Disable logging of network function calls.
EnableTrace	Enable logging of network function calls to a file.
EnumHostAliases	Enumerate the aliases for the specified host name or address.
EnumMailExchanges	Return a list of mail exchanges for the specified host name or IP address.
FormatAddress	Convert a numeric IPv4 address to a string.
GetAddress	Convert an address string in dotted notation to a numeric IPv4 address.
GetAddressFamily	Return the address family for the specified IP address.
GetDefaultHostFile	Return the full path to the default host file on the local system.
GetDefaultServicesFile	Return the full path to the default services file on the local system.
GetErrorString	Return a description for the specified error code.
GetHandle	Return the client handle used by this instance of the class.
GetHostAddress	Return the IP address of the specified hostname.
GetHostByAddress	Return a pointer to data for the specified host IP address.
GetHostByName	Return a pointer to data for the specified host name.
GetHostFile	Return the name of the current host file.
GetHostInfo	Return additional information for the specified host.
GetHostName	Return the host name for the specified IP address.
GetHostServices	Return a list of services supported by the specified host.
GetLastError	Return the last error code.
GetLocalAddress	Return the IP address for the local host.
GetLocalDomain	Return the local domain name for the current session.
GetLocalName	Return the local host name.
GetMailExchange	Return the host that processes mail for the specified domain.
GetRecord	Return record data for the current host.
GetResolverAddress	Return address of last nameserver that resolved query.

GetResolverOptions	Return the current resolver options for the client session.
GetRetryCount	Get the number of times the query is sent to each server.
GetServerAddress	Return the address of the specified nameserver.
GetServerPort	Return the port of the specified nameserver.
GetServiceName	Return the name of a service assigned to the specified port number.
GetServicePort	Return the port number assigned to the specified service name.
GetTimeout	Get the number of seconds until a query times out.
HostNameToUnicode	Converts the canonical form of a host name to its Unicode version.
IsInitialized	Determine if the class has been successfully initialized.
MatchHostName	Match a host name against of list of addresses including wildcards.
NormalizeHostName	Return the canonical form of a host name.
RegisterServer	Add a nameserver address to the current session.
Reset	Reset the current client state.
Resolve	Resolve a host name into an IP address or an IP address into a host name.
SetHostFile	Specify the name of an alternate file to use when resolving host names and IP addresses.
SetLastError	Set the last error code.
SetLocalDomain	Set the local domain name for the current session.
SetResolverOptions	Set the resolver options for the client session.
SetRetryCount	Set the number of times the query is sent to each server.
SetTimeout	Set the number of seconds until a query times out.
ShowError	Display a message box with a description of the specified error.
UnregisterServer	Remove a nameserver address from the current session.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::CDnsClient Method

```
CDnsClient();
```

The **CDnsClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **DnsInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

~CDnsClient, IsInitialized, SetResolverOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::~CDnsClient

```
~CDnsClient();
```

The **CDnsClient** destructor releases resources allocated by the current instance of the CDnsClient object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CDnsClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CDnsClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the class. A value of zero specifies that the current thread should become the owner of the class instance.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

AttachHandle, Cancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding queries initiated by the client.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

GetHostByAddress, GetHostByName, GetHostInfo, GetHostServices, GetMailExchange, GetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::EnumHostAliases Method

```
INT EnumHostAliases(
   LPCTSTR lpszHostName,
   LPTSTR *lpszHostAlias,
   INT nMaxAliases
);
```

The **EnumHostAliases** method returns a list of aliases for the specified host name or IP address.

Parameters

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used. This value may also be an IP address.

lpszHostAlias

Pointer to an array of string pointers which specify one or more host aliases. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved.

nMaxAliases

The maximum number of aliases in the array. This parameter must have a value of at least one, or an error will be returned.

Return Value

If the method succeeds, the return value is the number of host aliases. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify the host aliases or delete any of the values. This method uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::EnumMailExchanges Method

```
INT EnumMailExchanges(
   LPCTSTR lpszHostName,
   LPTSTR *lpszMailExchanges,
   INT nMaxMailExchanges
);
```

The **EnumMailExchanges** method returns a list of mail exchanges for the specified host name or IP address.

Parameters

lpszHostName

Pointer to the string that contains the hostname or domain name to be queried.

lpszMailExchanges

Pointer to an array of string pointers which specify one or more mail exchanges. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved. The list of mail exchange records is sorted in priority order, from highest (i.e., those whose preference value is smallest) to lowest.

nMaxMailExchanges

The maximum number of mail exchanges in the array. If this parameter is 0, then the method will return the number of mail exchanges, but the list of mail exchanges will not be output in *lpszMailExchanges*.

Return Value

If the method succeeds, the return value is the number of mail exchanges. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify the mail exchange host names or delete any of the values. This method uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other method calls.

Example

```
// Get the number of mail exchanges
if ((nMX = pClient->EnumMailExchanges(szHostName, NULL, 0)) == DNS_ERROR)
{
    pClient->ShowError();
}
else
{
    // Allocate memory for the list of mail exchanges
    lpszMailExchanges = (LPTSTR *)LocalAlloc(LPTR, (nMX * sizeof(LPTSTR)));

    // Retrieve the list of mail exchanges
    nMX = pClient->EnumMailExchanges(szHostName, lpszMailExchanges, nMX);

    // Populate a listbox with the mail exchanges
    for (int nIndex = 0; nIndex < nMX; nIndex++)
         pListBox->AddString(*lpszMailExchanges++);
```

```
        LocalFree((HLOCAL)lpszMailExchanges);
    }
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMailExchange

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::FormatAddress Method

```
INT FormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT FormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   CString& strAddress
);
```

The **FormatAddress** method converts a numeric IPv4 or IPv6 address to a string.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IPv4 or IPv6 address to be converted into a string.

lpszAddress

A null-terminated string which will contain the converted IPv4 address in dot-notation. This string should be at least 16 characters in length. If the Microsoft Foundation Classes are being used, the second form of this method may be called where a **CString** object is specified instead.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the method succeeds, the return value is the length of the string buffer. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetHostAddress, GetHostInfo, GetHostServices, GetMailExchange, GetRecord, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetAddress Method

```
INT GetAddress(
   LPCTSTR lpszAddress,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **GetAddress** method converts an address string to a numeric IPv4 or IPv6 address.

Parameters

lpszAddress

A null-terminated string which specifies an IPv4 address in dotted notation.

nAddressFamily

An integer value which specifies the type of IP address. If this parameter is zero, the address family will be determined automatically based on the format of the address string. If this parameter is DNS_ADDRESS_IPV4, the address must be in IPv4 format, and if it is DNS_ADDRESS_IPV6, the address must be in IPv6 format.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the numeric form of the IPv4 or IPv6 address in network byte order when the method returns.

Return Value

If the method succeeds, the return value is the address family for the IP address. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will only accept a string that is in the proper format for an IP address, and cannot be used to resolve a host name. To perform host name resolution, use the **GetHostAddress** method. To convert a numeric address to an address string, use the **FormatAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, GetHostAddress, GetHostInfo, GetHostServices, GetMailExchange, GetRecord, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetAddressFamily Method

```
INT GetAddressFamily(
   LPCTSTR lpszAddress
);
```

The **GetAddressFamily** method returns the address family for the specified IP address.

Parameters

lpszAddress

A null-terminated string which specifies an IPv4 or IPv6 address.

Return Value

If the method succeeds, the return value is DNS_ADDRESS_IPV4 if the address is in IPv4 format, or DNS_ADDRESS_IPV6 if the address is in IPv6 format. If the address string is not in a recognized format, it returns DNS_ADDRESS_UNKNOWN.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetDefaultHostFile Method

```
INT GetDefaultHostFile(
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetDefaultHostFile(
   CString& strFileName
);
```

The **GetDefaultHostFile** method returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostName** methods. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null byte. An alternative version of this method accepts a **CString** object.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **GetLastError**.

Remarks

This method returns the location of the host file and does not determine if the file actually exists. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname. To specify an alternate local host file, use the **SetHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultServicesFile, GetHostAddress, GetHostFile, GetHostName, SetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetDefaultServicesFile Method

```
INT GetDefaultServicesFile(
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetDefaultServicesFile(
   CString& strFileName
);
```

The **GetDefaultServicesFile** method returns the fully qualified path name of the services file on the local system. The services file is used as a database that maps a port number to a service name, and is used by the **GetServiceName** and **GetServicePort** methods. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
name port/protocol [alias ...]
```

For example, one typical entry maps the service name "http" to port 80, the standard port for the Hypertext Transfer Protocol. This would be entered as:

```
http 80/tcp www www-http
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified path to the default services file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte. An alternative version of this method accepts a **CString** object.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. A return value of zero indicates that the default services file could not be determined for the current platform. To get extended error information, call **GetLastError**.

Remarks

This method returns the location of the services file and does not determine if the file actually exists. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetServiceName, GetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostAddress Method

```
INT GetHostAddress(
   LPCTSTR lpszHostName,
   INT nAddressFamily,
   LPTSTR lpszHostAddress,
   INT nMaxLength
);

INT GetHostAddress(
   LPCTSTR lpszHostName,
   LPTSTR lpszHostAddress,
   INT nMaxLength
);

INT GetHostAddress(
   LPCTSTR lpszHostName,
   INT nAddressFamily,
   CString& strHostAddress
);

INT GetHostAddress(
   LPCTSTR lpszHostName,
   CString& strHostAddress
);
```

The **GetHostAddress** method resolves the specified host name, storing the IP address in the provided buffer.

Parameters

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of DNS_ADDRESS_IPV4 specifies that the IPv4 address for the host should be returned. A value of DNS_ADDRESS_IPV6 specifies that the IPv6 address for the host should be returned. A value of DNS_ADDRESS_ANY specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host. There are alternate versions of this method which omits the address family, in which case it will default to returning the IPv4 address for the host.

lpszHostAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 48 characters in length. The format of the address is determined by the address family specified.

nMaxLength

The maximum length of the string buffer. The maximum length of the buffer must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the host address buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call

GetLastError.

Remarks

The **GetHostAddress** method may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the ***nAddressFamily*** parameter as DNS_ADDRESS_IPV4 to prevent the possibility of an IPv6 address being returned.

If the ***nAddressFamily*** parameter is specified as DNS_ADDRESS_ANY, this function will first check for an IPv4 address record for the host. If it exists, it will return that address. If the host does not have an IPv4 address, it will then check for an IPv6 address record and return that address. This gives preference to IPv4 addresses, but your application should never depend on this behavior. In the future, this function may change to give preference to IPv6 addresses.

To determine what format an address is in, use the **GetAddressFamily** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHostAliases, GetAddressFamily GetHostName, GetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostByAddress Method

```
LPHOSTENT GetHostByAddress(
   LPVOID lpvAddress,
   INT cbAddress,
   INT nAddressFamily
);
```

The **GetHostByAddress** method returns a pointer to a HOSTENT structure which contains the results of a successful search for the host specified by address parameter.

Parameters

lpvAddress

Pointer to an integer IPv4 address in network byte order.

cbAddress

The length of the address in bytes; this value should always be 4.

nAddressFamily

The type of address being resolved; this value should always be DNS_ADDRESS_IPV4 as defined in the Windows Sockets header file.

Return Value

If the method succeeds, the return value is a pointer to a HOSTENT structure. If the method fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. To convert an IPv4 address string in dotted notation to a 32-bit IP address, use the **GetAddress** method.

This method is included for compatibility with existing applications which already use the HOSTENT structure. Because this method returns a pointer to a complex structure, it may not be suitable for some programming languages. There is no Unicode version of the HOSTENT structure and members will return pointers to ANSI strings even if the application is compiled to use Unicode.

This method is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **GetHostAddress** and **GetHostName** methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostByName, GetHostInfo, GetHostName, GetHostServices, GetRecord, GetResolverAddress, RegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostByName Method

```
LPHOSTENT GetHostByName(
   LPCTSTR lpszHostName
);
```

The **GetHostByName** method returns a pointer to a HOSTENT structure which contains the results of a successful search for the host specified in the name parameter.

Parameters

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

Return Value

If the method succeeds, the return value is a pointer to a HOSTENT structure. If the method fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. This method will automatically resolve an IP address passed as a string, converting it to numeric form and calling the **GetHostByAddress** method.

This method is included for compatibility with existing applications which already use the HOSTENT structure. Because this function returns a pointer to a complex structure, it may not be suitable for some programming languages. There is no Unicode version of the HOSTENT structure and members will return pointers to ANSI strings even if the application is compiled to use Unicode.

This method is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **GetHostAddress** and **GetHostName** methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostByAddress, GetHostInfo, GetHostName, GetHostServices, GetMailExchange, GetRecord, GetResolverAddress, RegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostFile Method

```
INT GetHostFile(
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetHostFile(
   CString& strFileName
);
```

The **GetHostFile** method returns the name of the host file previously set using the **SetHostFile** method. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostName** method.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the method was unable to determine the file name. To get extended error information, call **GetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the method failed.

Remarks

This method only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **SetHostFile** method.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this method may be different than the default host file for the local system. To determine the file name for the default host file, use the **GetDefaultHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetHostAddress, GetHostName, SetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostInfo Method

```
INT GetHostInfo(
   LPCTSTR lpszHostName,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **GetHostInfo** method returns the HINFO record for the specified hostname. This information, if it is provided, typically specifies the operating system type and hardware platform.

Parameters

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the host information buffer is returned. A return value of zero indicates that no information is available for the specified host. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

Many systems to do not maintain HINFO records for a site since that information can potentially be used to compromise system security. The information is typically used for administrative purposes with internal networks.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostByAddress, GetHostByName, GetHostServices, GetMailExchange, GetRecord, GetResolverAddress, RegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostName Method

```
INT GetHostName(
   LPCTSTR lpszHostAddress,
   LPTSTR lpszHostName,
   INT nMaxLength
);

INT GetHostName(
   LPCTSTR lpszHostAddress,
   CString& strHostName
);
```

The **GetHostName** method resolves the specified IP address, storing the fully qualified host name in the provided buffer.

Parameters

lpszHostAddress

Pointer to a string that specifies an IPv4 or IPv6 formatted address.

lpszHostName

Pointer to the buffer that will contain the fully qualified domain name for the specified host. This buffer should be at least 64 characters in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the host name buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetHostName** method first looks to see if there is an entry in the local host file for the specified IP address, and if one exists, it will return the host name for that address. If you do not want to use the local host file at all, and only return an host name if a DNS query resolves the address, use the **GetRecord** method and specify a record type of DNS_RECORD_PTR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHostAliases, GetHostByAddress, GetHostByName, GetHostAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetHostServices Method

```
INT GetHostServices(
   LPCTSTR lpszHostName,
   INT nProtocol,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **GetHostServices** method returns the WKS (Well Known Services) record for the specified hostname and protocol. This information, if it is provided, typically specifies the names of those services supported on the host.

Parameters

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nProtocol

The protocol for those services that information should be returned about. The following protocols are recognized:

Value	Description
INET_PROTOCOL_TCP	Services that use the Transmission Control Protocol (TCP)
INET_PROTOCOL_UDP	Services that use the User Datagram Protocol (UDP)

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the host services buffer is returned. A return value of zero indicates that no information is available for the specified host. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

Many systems to do not maintain complete services records for a site since that information can potentially be used to compromise system security. An application should not depend on this information being available for any given record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostByAddress, GetHostByName, GetHostInfo, GetMailExchange, GetRecord, GetResolverAddress,

RegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetLocalAddress Method

```
INT GetLocalAddress(
   INT nAddressFamily,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **DnsGetLocalAddress** method returns the IP address for the local host.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of DNS_ADDRESS_IPV4 specifies that the IPv4 address for the host should be returned. A value of DNS_ADDRESS_IPV6 specifies that the IPv6 address for the host should be returned. A value of DNS_ADDRESS_ANY specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host.

lpszAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 40 characters in length to accommodate both IPv4 and IPv6 addresses.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the host address buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **GetLocalAddress** method may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the ***nAddressFamily*** parameter as DNS_ADDRESS_IPV4 to prevent the possibility of an IPv6 address being returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostName, GetHostInfo, GetHostServices, GetMailExchange, GetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetLocalDomain Method

```
INT GetLocalDomain(
   LPTSTR lpszDomain,
   INT nMaxLength
);
```

The **GetLocalDomain** method copies the local domain name into the specified buffer. The value returned is the same value that was set with the **SetLocalDomain** method. If no local domain name has been set, an empty string is returned.

Parameters

lpszDomain

Pointer to the buffer that is used to store the local domain name. If no local domain name has been set, this buffer will be set to a zero-length string.

nMaxLength

The maximum number of bytes to copy into the buffer, including the null character terminator.

Return Value

If the method succeeds, the return value is the length of the domain name string. A return value of zero indicates that no local domain name has been set. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RegisterServer, SetLocalDomain

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetLocalName Method

```
INT GetLocalName(
   LPTSTR lpszLocalName,
   INT nMaxLength
);
```

The **GetLocalName** method returns the local host name.

Parameters

lpszLocalName

Pointer to a string buffer that will contain the local host name. It is recommended that this buffer be at least 64 characters in size.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is the length of the local host name. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostByAddress, GetHostByName, GetHostInfo, GetHostServices, GetMailExchange, GetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetMailExchange Method

```
INT GetMailExchange(
   LPCTSTR lpszHostName,
   LPINT lpnPreference,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **GetMailExchange** method returns the mail exchange (MX) record information for the specified domain. This information, if it is provided, identifies a server responsible for processing mail for the given domain.

Parameters

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpnPreference

Pointer to the integer which will contain the preference for the specified mail exchange host.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the buffer is returned. A return value of zero indicates that no information is available for the specified host. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The mail exchange record is typically used by mail delivery agents to determine what system is responsible for accepting mail addressed to a given domain. This method will return the first MX record provided by the server. Note that some domains may have multiple mail servers. To enumerate all of the mail exchange records for a domain, use the **EnumMailExchanges** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostByAddress, GetHostByName, GetHostInfo, GetHostServices, GetRecord, GetResolverAddress, RegisterServer, EnumMailExchanges

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetRecord Method

```
INT GetRecord(
   LPCTSTR lpszHostName,
   INT nRecordType,
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT GetRecord(
   LPCTSTR lpszHostName,
   INT nRecordType,
   CString& strBuffer
);
```

The **GetRecord** method returns the specified record information for the given hostname.

Parameters

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nRecordType

The record type for the information that should be returned. The following record types are recognized:

Value	Description
DNS_RECORD_NONE	No record type
DNS_RECORD_ADDRESS	Host address
DNS_RECORD_NS	Authoritative nameserver
DNS_RECORD_CNAME	Canonical name (alias)
DNS_RECORD_SOA	Start of Authority
DNS_RECORD_WKS	Well known services
DNS_RECORD_PTR	Domain name
DNS_RECORD_HINFO	Host information
DNS_RECORD_MINFO	Mailbox information
DNS_RECORD_MX	Mail exchange host
DNS_RECORD_TXT	Text strings
DNS_RECORD_LOC	Location information
DNS_RECORD_UINFO	User information
DNS_RECORD_UID	User ID
DNS_RECORD_GID	Group ID

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the information buffer is returned. A return value of zero indicates that no information for that record is available for the specified host. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetRecord** method can be used to resolve a host name into an IP address using the record type DNS_RECORD_ADDRESS. It can also be used to perform a reverse lookup and resolve an IP address into a host name by using the record type DNS_RECORD_PTR.

To determine the host that serves as the primary or master DNS for a zone, the record name should be specified as the domain name (e.g.: microsoft.com) and the record type should be DNS_RECORD_SOA. The value returned will the fully qualified domain name for host.

Note that this method does not reference a local host file when resolving host names or addresses. If the record lookup fails, this method will return an error even if there's an entry for the host in the file that has been specified by a call to **SetHostFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, GetHostInfo, GetHostServices, GetMailExchange, GetResolverAddress, RegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetResolverAddress Method

```
INT GetResolverAddress(
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetResolverAddress(
   CString& strAddress
);
```

The **GetResolverAddress** returns the address of the nameserver that resolved the last query.

Parameters

lpszAddress

A null-terminated string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the method succeeds, the return value is the length of the address, not including the terminating null character. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostByAddress, GetHostByName, GetHostInfo, GetHostServices, GetMailExchange, GetRecord, RegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetResolverOptions Method

```
DWORD GetResolverOptions();
```

The **GetResolverOptions** method returns the options that have been set for the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the resolver options set for the client session. If the client handle is invalid or no resolver options have been specified, the function will return zero.

Remarks

The **GetResolverOptions** method can be used to determine which resolver options have been specified for the client session. For a list of the available options, refer to the documentation for the **SetResolverOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

CDnsClient, SetResolverOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetRetryCount Method

```
INT GetRetryCount();
```

The **GetRetryCount** returns the retry count for the current client session.

Return Value

If the method succeeds, the return value is the retry count. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetTimeout, SetRetryCount, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetServerAddress Method

```
INT GetServerAddress(
   INT nServer,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetServerAddress(
   INT nServer,
   CString& strAddress
);
```

The **GetServerAddress** method returns the address of the specified nameserver.

Parameters

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **RegisterServer** method when the nameserver is registered.

lpszAddress

A null-terminated string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the method succeeds, the return value is the length of the address, not including the terminating null character. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResolverAddress, GetServerPort, RegisterServer, UnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetServerPort Method

```
INT GetServerPort(
   INT nServer
);
```

The **GetServerPort** method returns the port number registered to the specified nameserver.

Parameters

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **RegisterServer** method when the nameserver is registered.

Return Value

If the method succeeds, the return value is the specified port number. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResolverAddress, GetServerAddress, RegisterServer, UnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetServiceName Method

```
INT GetServiceName(
   INT nServicePort,
   INT nServiceType,
   LPTSTR lpszServiceName,
   INT nMaxLength
);

INT GetServiceName(
   INT nServicePort,
   INT nServiceType,
   CString& strServiceName
);
```

The **GetServiceName** method returns the service name assigned to the specified port number.

Parameters

nServicePort

An integer which specifies the port number. This value must be a valid port number in the range of 1 through 65535.

nServiceType

An integer which specifies the protocol associated with the service. This may be one of the following values:

Value	Description
DNS_SERVICE_ANY	Match the first service entry for the specified port number regardless of the protocol used.
DNS_SERVICE_TCP	Match a service entry for the specified port number which uses the TCP protocol.
DNS_SERVICE_UDP	Match a service entry for the specified port number which uses the UDP protocol.

lpszServiceName

A null-terminated string buffer which will contain the name of the service assigned to the port number when the method returns. This parameter cannot be NULL and the buffer must be large enough to store the complete service name, including the terminating null character. An alternative version of this method accepts a **CString** object.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Port numbers can be assigned human readable service names to make them easier to reference. This

method will return the name assigned to the specified port number. If the port number does not have an assigned name, this method will return zero and the last error code will be set to ST_ERROR_INVALID_SERVICE_NAME.

Most applications should use DNS_SERVICE_ANY to return the service name assigned to a port number. Services typically use the same name and port number regardless of the protocol used to establish a connection with them. However, specifying a protocol can enable your application to determine if the service supports the protocol. For example, calling this method with port number 21 and DNS_SERVICE_UDP would result in an error because FTP only supports TCP connections.

Service names are assigned by the Internet Assigned Numbers Authority (IANA) and the database of recognized service names can be returned by calling the **GetDefaultServicesFile** method. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetDefaultServicesFile, GetHostFile, GetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetServicePort Method

```
INT GetServicePort(
   LPCTSTR lpszServiceName,
   INT nServiceType
);
```

The **GetServicePort** method returns the port number assigned to the specified service.

Parameters

lpszServiceName

A null-terminated string which specifies the service name. Service names are not case sensitive.

nServiceType

An integer which specifies the protocol associated with the service. This may be one of the following values:

Value	Description
DNS_SERVICE_ANY	Match the first service entry for the specified port number regardless of the protocol used.
DNS_SERVICE_TCP	Match a service entry for the specified port number which uses the TCP protocol.
DNS_SERVICE_UDP	Match a service entry for the specified port number which uses the UDP protocol.

Return Value

If the method succeeds, the return value is the port number assigned to the service name. If the method fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

Port numbers can be assigned human readable service names to make them easier to reference. This method will return the port number assigned to the specified service name. If the service name does not exist, this method will return DNS_ERROR and the last error code will be set to ST_ERROR_INVALID_SERVICE_NAME.

Most applications should use DNS_SERVICE_ANY to return the port number assigned to a service. Services typically use the same name and port number regardless of the protocol used to establish a connection with them. However, specifying a protocol can enable your application to determine if the service supports the protocol. For example, calling this method with the service name `"ftp"` and DNS_SERVICE_UDP would result in an error because FTP only supports TCP connections.

Service names are assigned by the Internet Assigned Numbers Authority (IANA) and the database of recognized service names can be returned by calling the **GetDefaultServicesFile** method. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetDefaultServicesFile, GetHostFile, GetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the timeout value for the current client session.

Return Value

If the method succeeds, the return value is the timeout period. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

GetRetryCount, SetRetryCount, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::HostNameToUnicode Method

```
INT HostNameToUnicode(
   LPCTSTR lpszHostName,
   LPTSTR lpszUnicodeName,
   INT nMaxLength
);

INT HostNameToUnicode(
   LPCTSTR lpszHostName,
   CString& strUnicodeName
);
```

The **HostNameToUnicode** method converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a CString object.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **HostNameToUnicode** method will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this method, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the application is compiled using the Unicode character set, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI character set is used, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the GetHostAddress method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, NormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

CDnsClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::MatchHostName Method

```
BOOL MatchHostName(
   LPCTSTR lpszHostName,
   LPCTSTR lpszHostMask
   BOOL bResolve
);
```

The **MatchHostName** method matches a host name against one more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetHostAddress, GetHostName, GetLocalAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::NormalizeHostName Method

```
INT NormalizeHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszNormalized,
   INT nMaxLength
);

INT NormalizeHostName(
   LPCTSTR lpszHostName,
   CString& strNormalized
);
```

The **NormalizeHostName** method returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a CString object.

nMaxLength

The maximum number of characters that can be copied to the ***lpszNormalized*** string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **NormalizeHostName** method will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this method, it will be converted to an ASCII compatible format for domain names.

If the application is compiled using the Unicode character set, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended that you use Unicode.

Although this method performs checks to ensure that the ***lpszHostName*** parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the GetHostAddress method.

It is recommended that you use this method if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this method prior to calling the other methods that accept a host name as a parameter. They already normalize the host name and perform checks to ensure it is in the correct format.

If the *lpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the GetHostName method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, HostNameToUnicode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::RegisterServer Method

```
INT RegisterServer(
   INT nServer,
   LPCTSTR lpszHostAddress,
   INT nPort
);
```

The **RegisterServer** method registers a nameserver with the current client session. The nameserver is used to resolve queries issued by the client, such as returning the IP address for a given host name. At least one nameserver must be registered by the client before queries are issued.

Parameters

nServer

The index into the client nameserver table. This index, starting at 0, is used to specify which slot in the client's nameserver table will be used to store the nameserver information.

lpszHostAddress

A null-terminated string that specifies the IP address of the nameserver to be registered. Note that hostnames cannot be specified.

nPort

The port number that the specified nameserver is accepting queries on. This value may be set to zero, in which case it will use the default port value.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetServerAddress, GetServerPort, UnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the current state of the client session. The timeout and retry counts are set to their default values, the local domain name is cleared and all registered servers are removed from the client nameserver table.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RegisterServer, SetLocalDomain, SetRetryCount, SetTimeout, UnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::Resolve Method

```
BOOL Resolve(
   LPCTSTR lpszRecord,
   LPTSTR lpszResult,
   INT nMaxLength,
   BOOL bHostName
);

BOOL Resolve(
   LPCTSTR lpszRecord,
   LPINTERNET_ADDRESS lpAddress
);

BOOL Resolve(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszHostName,
   INT nMaxLength
);

BOOL Resolve(
   LPCTSTR lpszRecord,
   CString& strResult,
   BOOL bHostName
);

BOOL Resolve(
   LPINTERNET_ADDRESS lpAddress,
   CString& strHostName
);
```

The **Resolve** method resolves the specified host name into an IP address, or an IP address into its corresponding host name.

Parameters

lpszRecord

Pointer to a string which specifies the record to be resolved. If the *bHostName* argument is non-zero, then the record is expected to be a host name; otherwise it is expected to be an IP address string in dot notation.

lpszResult

Pointer to the buffer that will contain the result of the nameserver query. If the *bHostName* argument is non-zero, then the string will contain the host's IP address in dot notation when the method returns; otherwise, it will contain the host name for the IP address specified in the *lpszRecord* argument.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

bHostName

A boolean argument which determines how the record is resolved. If this value is non-zero, then the *lpszRecord* argument should specify a host name and the *lpszResult* buffer will contain the IP address for that host when the method returns. If this value is zero, then the *lpszRecord* argument should specify an IP address and the *lpszResult* buffer will contain its host name when the method returns.

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric form of an IPv4 or IPv6 address in network byte order. This can be used with versions of the method that resolve host names and addresses using the numeric form of the IP address instead of a string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resolve** method provides a convenient interface to resolve host names and IP addresses. Note that this method first looks to see if there is an entry in the local host file for the specified host name or IP address, and if one exists, it will return that record without querying the name server.

Example

```
// Create an instance of the class object
CDnsClient dnsClient;

// Resolve the address of the www.microsoft.com server
CString strHostName = _T("www.microsoft.com");
INTERNET_ADDRESS ipHostAddress;

// If the host name is resolved, then perform a reverse
// lookup on that address to obtain the actual host name
// for that server
if (dnsClient.Resolve(strHostName, &ipHostAddress))
    dnsClient.Resolve(&ipHostAddress, strHostName);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, FormatAddress, GetHostAddress, GetHostFile, GetHostName, SetHostFile, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::SetHostFile Method

```
INT SetHostFile(
   LPCTSTR lpszFileName
);
```

The **SetHostFile** method specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostName** methods. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the method succeeds, the return value is the number of entries in the host file. A return value of DNS_ERROR indicates failure. To get extended error information, call **GetLastError**.

Remarks

This method loads the file into memory allocated for the current thread. If the contents of the file have changed after the method has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this method again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **GetHostFile** method. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **GetDefaultHostFile** method. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetHostAddress, GetHostFile, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the error code for the current thread. This method is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread. A value of zero clears the last error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::SetLocalDomain Method

```
INT SetLocalDomain(
   LPCTSTR lpszDomain
);
```

Parameters

lpszDomain

Pointer to the string which contains the local domain name. This is used as a default value when a query does not explicitly specify a domain name.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLocalDomain, GetRetryCount, GetTimeout, Reset, SetRetryCount, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::SetResolverOptions Method

```
DWORD SetResolverOptions(
   DWORD dwOptions
);
```

The **SetResolverOptions** method changes the resolver options for the client session.

Parameters

hClient

Handle to the client session.

dwOptions

An unsigned integer that specifies one or more resolver options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
DNS_OPTION_NONE	No additional resolver options specified. This is the default value, and it is recommended that most applications do not specify additional options unless the implications of doing so are understood.
DNS_OPTION_PROMOTE	Promotes the server that successfully completed the last query to the first server that will be used to resolve subsequent queries. This option can improve performance in some cases where one or more of the registered servers are non-responsive. This option takes precedence over the DNS_OPTION_ROTATE option.
DNS_OPTION_ROTATE	Enables a round-robin selection of nameservers when performing queries. Normally each nameserver is queried in the same order. This option rotates the available nameservers so a different server is used with each query.
DNS_OPTION_AUTHONLY	Require the answer from the nameserver to be authoritative, not from the server's cache. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_PRIMARY	Queries are only accepted from the primary nameserver. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_NORECURSE	Disable the sending of recursive queries to the nameserver. Specifying this option will disable the bit in the DNS request header that specifies recursion is desired.
DNS_OPTION_NOSEARCH	Disable additional queries of higher domains in the search list if the host name cannot be resolved. If this option is specified, and the host name cannot be resolved using the local domain name an error is returned immediately. This option is ignored if no local

	domain has been specified or if the DNS_OPTION_NOSUFFIX option has been specified.
DNS_OPTION_NOSUFFIX	Disable additional queries using the local domain name if the host name is not a fully qualified domain name and cannot be resolved. This option is ignored if no local domain has been specified.
DNS_OPTION_NONAMECHECK	Disable checking the host name for invalid characters, such as the underscore and control characters. By default, host names are checked to ensure they're valid before submitting a query to the nameserver.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetResolverOptions** method changes the resolver options for the specified client session, modifying how nameserver queries are processed. It is recommended that most applications do not specify any resolver options and use the default behavior. Specifying these options without understanding how they can affect standard queries can result in unexpected failures. In particular, caution should be used when specifying the DNS_OPTION_NORECURSE and DNS_OPTION_NOSEARCH options as they change the normal process of resolving a host name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

CDnsClient, GetResolverOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::SetRetryCount Method

```
INT SetRetryCount(
   INT nRetries
);
```

The **SetRetryCount** method sets the number of attempts that the client will make attempting to resolve a query. When used in conjunction with the **SetTimeout** method, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

nRetries

The number of attempts the client will make, per nameserver, to resolve a query.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetRetryCount, GetTimeout, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::SetTimeout Method

```
INT SetTimeout(
   INT nTimeout
);
```

The **SetTimeout** method sets the number of seconds that the client will wait for a response from a nameserver. The timeout value is used each time a server in the client's nameserver table is queried. When used in conjunction with the **SetRetryCount** method, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

nTimeout

The number of seconds until the client times out waiting for a response from a nameserver.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, with two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

GetRetryCount, GetTimeout, SetRetryCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** function in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CDnsClient::UnregisterServer Method

```
INT UnregisterServer(
   INT nServer
);
```

The **UnregisterServer** method removes the specified nameserver information from the client. Unregistering a server prevents the client from using that server to satisfy subsequent DNS queries.

Parameters

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **RegisterServer** method when the nameserver is registered.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetServerAddress, GetServerPort, RegisterServer, Reset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Data Structures

- HOSTENT
- INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HOSTENT

The **HOSTENT** structure contains information about a domain name, including its IP address and any aliases which may be defined.

```
typedef struct _HOSTENT
{
   char *  h_name;
   char ** h_aliases;
   short   h_addrtype;
   short   h_length;
   char ** h_addr_list;
} HOSTENT, *LPHOSTENT;
```

Members

h_name

The fully qualified domain name (FQDN) that caused the nameserver server to return a reply.

h_aliases

A NULL-terminated array of alternate names.

h_addrtype

The type of address being returned.

h_length

The length, in bytes, of each address.

h_addr_list

A NULL-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h_addr** is defined to be **h_addr_list**[0] for compatibility with older software.

Remarks

This structure is intended to be used for backwards compatibility with legacy applications and should not be used with new applications. You cannot use this structure to obtain information for a host which is assigned an IPv6 address. Applications should use the **GetHostAddress** and **GetHostName** methods which support both IPv4 and IPv6 addresses.

The application must never attempt to modify this structure or free any of its members. There is no Unicode version of this structure, string pointers will always specify ANSI strings, even if the application is compiled to use Unicode. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before calling any other methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: winsock2.h.

See Also

GetHostAddress, GetHostByAddress, GetHostByName, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
     INT    ipFamily;
     BYTE   ipNumber[16];
}  INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Constants

Value	Description
DNS_RECORD_NONE	No record type
DNS_RECORD_ADDRESS	Host address
DNS_RECORD_NS	Authoritative nameserver
DNS_RECORD_CNAME	Canonical name (alias)
DNS_RECORD_SOA	Start of Authority
DNS_RECORD_WKS	Well known services
DNS_RECORD_PTR	Domain name
DNS_RECORD_HINFO	Host information
DNS_RECORD_MINFO	Mailbox information
DNS_RECORD_MX	Mail exchange host
DNS_RECORD_TXT	Text strings
DNS_RECORD_LOC	Location information
DNS_RECORD_UINFO	User information
DNS_RECORD_UID	User ID
DNS_RECORD_GID	Group ID

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketTools Library Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file type is invalid or not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified file type is invalid or not a regular file
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of function parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to function
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to function
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to function
0x80042727	ST_ERROR_BUFFER_TOO_SMALL	The specified buffer is too small

0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A network operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to function
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified
0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid or not supported
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	The specified protocol family is invalid or not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Network connection aborted by local host
0x80042746	ST_ERROR_CONNECTION_RESET	Network connection aborted by remote host
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host

0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	The specified host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes have initialized the networking subsystem
0x80042755	ST_ERROR_TOO_MANY_THREADS	Too many threads have been created by the current process
0x80042756	ST_ERROR_TOO_MANY_SESSIONS	Too many client sessions have been created by the current process
0x80042762	ST_ERROR_INTERNAL_FAILURE	An unexpected internal error has occurred
0x8004276B	ST_ERROR_NETWORK_NOT_READY	The networking subsystem is not available on this system
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence
0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The specified runtime license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This function has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified

0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open certificate store or establish security credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80042FA9	ST_ERROR_OPERATION_NOT_SUPPORTED	The specified operation is not supported
0x80042FAA	ST_ERROR_INVALID_PROTOCOL_VERSION	Invalid application protocol version specified
0x80042FAB	ST_ERROR_NO_SERVER_RESPONSE	No data returned from server
0x80042FAC	ST_ERROR_INVALID_SERVER_RESPONSE	Invalid data returned from server
0x80042FAD	ST_ERROR_UNEXPECTED_SERVER_RESPONSE	Unexpected response code returned from server
0x80042FAE	ST_ERROR_SERVER_TRANSACTION_FAILED	Server transaction failed
0x80042FAF	ST_ERROR_SERVICE_UNAVAILABLE	The service is currently unavailable
0x80042FB0	ST_ERROR_SERVICE_NOT_READY	The service is not ready, try again later
0x80042FB1	ST_ERROR_SERVER_RESYNC_FAILED	Unable to resynchronize with server
0x80042FB2	ST_ERROR_INVALID_PROXY_TYPE	Invalid proxy server type specified
0x80042FB3	ST_ERROR_PROXY_REQUIRED	Resource must be accessed through specified proxy
0x80042FB4	ST_ERROR_INVALID_PROXY_LOGIN	Unable to login to proxy server using specified credentials
0x80042FB5	ST_ERROR_PROXY_RESYNC_FAILED	Unable to resynchronize with proxy server
0x80042FB6	ST_ERROR_INVALID_COMMAND	Invalid command specified
0x80042FB7	ST_ERROR_INVALID_COMMAND_PARAMETER	Invalid command parameter specified
0x80042FB8	ST_ERROR_INVALID_COMMAND_SEQUENCE	Invalid command sequence specified
0x80042FB9	ST_ERROR_COMMAND_NOT_IMPLEMENTED	Specified command not implemented on this server

0x80042FBA	ST_ERROR_COMMAND_NOT_AUTHORIZED	Specified command not authorized for the current user
0x80042FBB	ST_ERROR_COMMAND_ABORTED	Specified command was aborted by the remote host
0x80042FBC	ST_ERROR_OPTION_NOT_SUPPORTED	The specified option is not supported on this server
0x80042FBD	ST_ERROR_REQUEST_NOT_COMPLETED	The current request has not been completed
0x80042FBE	ST_ERROR_INVALID_USERNAME	The specified username is invalid
0x80042FBF	ST_ERROR_INVALID_PASSWORD	The specified password is invalid
0x80042FC0	ST_ERROR_INVALID_ACCOUNT	The specified account name is invalid
0x80042FC1	ST_ERROR_ACCOUNT_REQUIRED	Account name has not been specified
0x80042FC2	ST_ERROR_INVALID_AUTHENTICATION_TYPE	Invalid or unsupported authentication protocol specified
0x80042FC3	ST_ERROR_AUTHENTICATION_REQUIRED	User authentication is required
0x80042FC4	ST_ERROR_PROXY_AUTHENTICATION_REQUIRED	Proxy authentication required
0x80042FC5	ST_ERROR_ALREADY_AUTHENTICATED	User has already been authenticated
0x80042FC6	ST_ERROR_AUTHENTICATION_FAILED	Unable to authenticate the specified user
0x80042FDB	ST_ERROR_NETWORK_ADAPTER	Unable to determine network adapter configuration
0x80042FDC	ST_ERROR_INVALID_RECORD_TYPE	Invalid record type specified
0x80042FDD	ST_ERROR_INVALID_RECORD_NAME	Invalid record name specified
0x80042FDE	ST_ERROR_INVALID_RECORD_DATA	Invalid record data specified
0x80042FDF	ST_ERROR_CONNECTION_OPEN	Data connection already established
0x80042FE0	ST_ERROR_CONNECTION_CLOSED	Server closed data connection
0x80042FE1	ST_ERROR_CONNECTION_PASSIVE	Data connection is passive
0x80042FE2	ST_ERROR_CONNECTION_FAILED	Unable to open data connection to server
0x80042FE3	ST_ERROR_INVALID_SECURITY_LEVEL	Data connection cannot be opened with this security setting
0x80042FE4	ST_ERROR_CACHED_TLS_REQUIRED	Data connection requires cached TLS session
0x80042FE5	ST_ERROR_DATA_READ_ONLY	Data connection is read-only
0x80042FE6	ST_ERROR_DATA_WRITE_ONLY	Data connection is write-only
0x80042FE7	ST_ERROR_END_OF_DATA	End of data
0x80042FE8	ST_ERROR_REMOTE_FILE_UNAVAILABLE	Remote file is unavailable
0x80042FE9	ST_ERROR_INSUFFICIENT_STORAGE	Insufficient storage on server
0x80042FEA	ST_ERROR_STORAGE_ALLOCATION	File exceeded storage allocation on

		server
0x80042FEB	ST_ERROR_DIRECTORY_EXISTS	The specified directory already exists
0x80042FEC	ST_ERROR_DIRECTORY_EMPTY	No files returned by the server for the specified directory
0x80042FED	ST_ERROR_END_OF_DIRECTORY	End of directory listing
0x80042FEE	ST_ERROR_UNKNOWN_DIRECTORY_FORMAT	Unknown directory format
0x80042FEF	ST_ERROR_INVALID_RESOURCE	Invalid resource name specified
0x80042FF0	ST_ERROR_RESOURCE_REDIRECTED	The specified resource has been redirected
0x80042FF1	ST_ERROR_RESOURCE_RESTRICTED	Access to this resource has been restricted
0x80042FF2	ST_ERROR_RESOURCE_NOT_MODIFIED	The specified resource has not been modified
0x80042FF3	ST_ERROR_RESOURCE_NOT_FOUND	The specified resource cannot be found
0x80042FF4	ST_ERROR_RESOURCE_CONFLICT	Request could not be completed due to the current state of the resource
0x80042FF5	ST_ERROR_RESOURCE_REMOVED	The specified resource has been permanently removed from this server
0x80042FF6	ST_ERROR_CONTENT_LENGTH_REQUIRED	Request must include the content length
0x80042FF7	ST_ERROR_REQUEST_PRECONDITION	Request could not be completed due to server precondition
0x80042FF8	ST_ERROR_UNSUPPORTED_MEDIA_TYPE	Request specified an unsupported media type
0x80042FF9	ST_ERROR_INVALID_CONTENT_RANGE	Content range specified for this resource is invalid
0x80042FFA	ST_ERROR_INVALID_MESSAGE_PART	Message is not multipart or an invalid message part was specified
0x80042FFB	ST_ERROR_INVALID_MESSAGE_HEADER	The specified message header is invalid or has not been defined
0x80042FFC	ST_ERROR_INVALID_MESSAGE_BOUNDARY	The multipart message boundary has not been defined
0x80042FFD	ST_ERROR_NO_FILE_ATTACHMENT	The current message part does not contain a file attachment
0x80042FFE	ST_ERROR_UNKNOWN_FILE_TYPE	The specified file type could not be determined
0x80042FFF	ST_ERROR_DATA_NOT_ENCODED	The specified data block could not be encoded
0x80043000	ST_ERROR_DATA_NOT_DECODED	The specified data block could not be decoded
0x80043001	ST_ERROR_FILE_NOT_ENCODED	The specified file could not be encoded

0x80043002	ST_ERROR_FILE_NOT_DECODED	The specified file could not be decoded
0x80043003	ST_ERROR_NO_MESSAGE_TEXT	The message does not contain any text characters
0x80043004	ST_ERROR_INVALID_MESSAGE_TEXT	The message contains invalid text characters
0x80043005	ST_ERROR_INVALID_ENCODING_TYPE	Invalid encoding type specified
0x80043006	ST_ERROR_INVALID_MESSAGE_NUMBER	Invalid message number specified
0x80043007	ST_ERROR_NO_RETURN_ADDRESS	No valid return address specified
0x80043008	ST_ERROR_NO_VALID_RECIPIENTS	No valid recipients specified
0x80043009	ST_ERROR_INVALID_RECIPIENT	The specified recipient address is invalid
0x8004300A	ST_ERROR_RELAY_NOT_AUTHORIZED	The specified domain is invalid or server will not relay messages
0x8004300B	ST_ERROR_MAILBOX_UNAVAILABLE	Specified mailbox is currently unavailable
0x8004300C	ST_ERROR_MAILBOX_READONLY	The selected mailbox cannot be modified
0x8004300D	ST_ERROR_MAILBOX_NOT_SELECTED	No mailbox has been selected
0x8004300E	ST_ERROR_INVALID_MAILBOX	Specified mailbox is invalid
0x8004300F	ST_ERROR_INVALID_DOMAIN	The specified domain name is invalid or not recognized
0x80043010	ST_ERROR_INVALID_SENDER	The specified sender address is invalid or not recognized
0x80043011	ST_ERROR_MESSAGE_NOT_DELIVERED	Message not delivered to any of the specified recipients
0x80043012	ST_ERROR_END_OF_MESSAGE_DATA	No more message data available to be read
0x80043013	ST_ERROR_INVALID_MESSAGE_SIZE	The specified message size is invalid
0x80043014	ST_ERROR_MESSAGE_NOT_CREATED	The message could not be created in the specified mailbox
0x80043015	ST_ERROR_NO_MORE_MAILBOXES	No more mailboxes exist on this server
0x80043016	ST_ERROR_INVALID_EMULATION_TYPE	The specified terminal emulation type is invalid
0x80043017	ST_ERROR_INVALID_FONT_HANDLE	The specified font handle is invalid
0x80043018	ST_ERROR_INVALID_FONT_NAME	The specified font name is invalid or unavailable
0x80043019	ST_ERROR_INVALID_PACKET_SIZE	The specified packet size is invalid
0x8004301A	ST_ERROR_INVALID_PACKET_DATA	The specified packet data is invalid
0x8004301B	ST_ERROR_INVALID_PACKET_ID	The unique packet identifier is invalid
0x8004301C	ST_ERROR_PACKET_TTL_EXPIRED	The specified packet time-to-live period has expired

0x8004301D	ST_ERROR_INVALID_NEWSGROUP	Invalid newsgroup specified
0x8004301E	ST_ERROR_NO_NEWSGROUP_SELECTED	No newsgroup selected
0x8004301F	ST_ERROR_EMPTY_NEWSGROUP	No articles in specified newsgroup
0x80043020	ST_ERROR_INVALID_ARTICLE	Invalid article number specified
0x80043021	ST_ERROR_NO_ARTICLE_SELECTED	No article selected in the current newsgroup
0x80043022	ST_ERROR_FIRST_ARTICLE	First article in current newsgroup
0x80043023	ST_ERROR_LAST_ARTICLE	Last article in current newsgroup
0x80043024	ST_ERROR_ARTICLE_EXISTS	Unable to transfer article, article already exists
0x80043025	ST_ERROR_ARTICLE_REJECTED	Unable to transfer article, article rejected
0x80043026	ST_ERROR_ARTICLE_TRANSFER_FAILED	Article transfer failed
0x80043027	ST_ERROR_ARTICLE_POSTING_DENIED	Posting is not permitted on this server
0x80043028	ST_ERROR_ARTICLE_POSTING_FAILED	Unable to post article to the server
0x80043029	ST_ERROR_INVALID_DATE_FORMAT	The specified date format is not recognized
0x8004302A	ST_ERROR_FEATURE_NOT_SUPPORTED	The specified feature is not supported
0x8004302B	ST_ERROR_INVALID_FORM_HANDLE	The specified form handle is invalid or a form has not been created
0x8004302C	ST_ERROR_INVALID_FORM_ACTION	The specified form action is invalid or has not been specified
0x8004302D	ST_ERROR_INVALID_FORM_METHOD	The specified form method is invalid or not supported
0x8004302E	ST_ERROR_INVALID_FORM_TYPE	The specified form type is invalid or not supported
0x8004302F	ST_ERROR_INVALID_FORM_FIELD	The specified form field name is invalid or does not exist
0x80043030	ST_ERROR_EMPTY_FORM	The specified form does not contain any field values
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated

		with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session
0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections
0x8004303A	ST_ERROR_NO_MESSAGE_STORE	No message store has been specified
0x8004303B	ST_ERROR_MESSAGE_STORE_CHANGED	The message store has changed since it was last accessed
0x8004303C	ST_ERROR_MESSAGE_NOT_FOUND	No message was found that matches the specified criteria
0x8004303D	ST_ERROR_MESSAGE_DELETED	The specified message has been deleted
0x8004303E	ST_ERROR_FILE_CHECKSUM_MISMATCH	The specified file checksums do not match
0x8004303F	ST_ERROR_FILE_SIZE_MISMATCH	The specified file sizes do not match
0x80043040	ST_ERROR_INVALID_FEED_URL	The news feed URL is invalid or specifies an unsupported protocol
0x80043041	ST_ERROR_INVALID_FEED_FORMAT	The internal format of the news feed is invalid
0x80043042	ST_ERROR_INVALID_FEED_VERSION	This version of the news feed is not supported
0x80043043	ST_ERROR_CHANNEL_EMPTY	There are no valid items found in this news feed
0x80043044	ST_ERROR_INVALID_ITEM_NUMBER	The specified channel item identifier is invalid
0x80043045	ST_ERROR_ITEM_NOT_FOUND	The specified channel item could not be found
0x80043046	ST_ERROR_ITEM_EMPTY	The specified channel item does not contain any data
0x80043047	ST_ERROR_INVALID_ITEM_PROPERTY	The specified item property name is invalid
0x80043048	ST_ERROR_ITEM_PROPERTY_NOT_FOUND	The specified item property has not been defined
0x80043049	ST_ERROR_INVALID_CHANNEL_TITLE	The channel title is invalid or has not been defined
0x8004304A	ST_ERROR_INVALID_CHANNEL_LINK	The channel hyperlink is invalid or has not been defined
0x8004304B	ST_ERROR_INVALID_CHANNEL_DESCRIPTION	The channel description is invalid or has not been defined

0x8004304C	ST_ERROR_INVALID_ITEM_TEXT	The description for an item is invalid or has not been defined
0x8004304D	ST_ERROR_INVALID_ITEM_LINK	The hyperlink for an item is invalid or has not been defined
0x8004304E	ST_ERROR_INVALID_SERVICE_TYPE	The specified service type is invalid
0x8004304F	ST_ERROR_SERVICE_SUSPENDED	Access to the specified service has been suspended
0x80043050	ST_ERROR_SERVICE_RESTRICTED	Access to the specified service has been restricted
0x80043051	ST_ERROR_INVALID_PROVIDER_NAME	The specified provider name is invalid or unknown
0x80043052	ST_ERROR_INVALID_PHONE_NUMBER	The specified phone number is invalid or not supported in this region
0x80043053	ST_ERROR_GATEWAY_NOT_FOUND	A message gateway cannot be found for the specified provider
0x80043054	ST_ERROR_MESSAGE_TOO_LONG	The message exceeds the maximum number of characters permitted
0x80043055	ST_ERROR_INVALID_PROVIDER_DATA	The request returned invalid or incomplete service provider data
0x80043056	ST_ERROR_INVALID_GATEWAY_DATA	The request returned invalid or incomplete message gateway data
0x80043057	ST_ERROR_MULTIPLE_PROVIDERS	The request has returned multiple service providers
0x80043058	ST_ERROR_PROVIDER_NOT_FOUND	The specified service provider could not be found
0x80043059	ST_ERROR_INVALID_MESSAGE_SERVICE	The specified message is not supported with this service type
0x8004305A	ST_ERROR_INVALID_MESSAGE_FORMAT	The specified message format is invalid
0x8004305B	ST_ERROR_INVALID_CONFIGURATION	One or more configuration options are invalid for this service
0x8004305C	ST_ERROR_SERVER_ACTIVE	The requested action is not permitted while the server is active
0x8004305D	ST_ERROR_SERVER_PORT_BOUND	Unable to obtain exclusive use of the specified local port
0x8004305E	ST_ERROR_INVALID_CLIENT_SESSION	The specified client identifier is invalid for this session
0x8004305F	ST_ERROR_CLIENT_NOT_IDENTIFIED	The specified client has not provided user credentials
0x80043060	ST_ERROR_INVALID_CLIENT_STATE	The requested action cannot be performed at this time
0x80043061	ST_ERROR_INVALID_RESULT_CODE	The specified result code is not valid for

		this protocol
0x80043062	ST_ERROR_COMMAND_REQUIRED	The specified command is required and cannot be disabled
0x80043063	ST_ERROR_COMMAND_DISABLED	The specified command has been disabled
0x80043064	ST_ERROR_COMMAND_SEQUENCE	The command cannot be processed at this time
0x80043065	ST_ERROR_COMMAND_COMPLETED	The previous command has completed
0x80043066	ST_ERROR_INVALID_PROGRAM_NAME	The specified program name is invalid or unrecognized
0x80043067	ST_ERROR_INVALID_REQUEST_HEADER	The request header contains one or more invalid values
0x80043068	ST_ERROR_INVALID_VIRTUAL_HOST	The specified virtual host name is invalid
0x80043069	ST_ERROR_VIRTUAL_HOST_NOT_FOUND	The specified virtual host does not exist
0x8004306A	ST_ERROR_TOO_MANY_VIRTUAL_HOSTS	Too many virtual hosts created for this server
0x8004306B	ST_ERROR_INVALID_VIRTUAL_PATH	The specified virtual path name is invalid
0x8004306C	ST_ERROR_VIRTUAL_PATH_NOT_FOUND	The specified virtual path does not exist
0x8004306D	ST_ERROR_TOO_MANY_VIRTUAL_PATHS	Too many virtual paths created for this server
0x8004306E	ST_ERROR_INVALID_TASK	The asynchronous task identifier is invalid
0x8004306F	ST_ERROR_TASK_ACTIVE	The asynchronous task has not finished
0x80043070	ST_ERROR_TASK_QUEUED	The asynchronous task has been queued
0x80043071	ST_ERROR_TASK_SUSPENDED	The asynchronous task has been suspended
0x80043072	ST_ERROR_TASK_FINISHED	The asynchronous task has finished
0x80043073	ST_ERROR_INVALID_ACCOUNT_UUID	The account unique identifier is invalid
0x80043074	ST_ERROR_INVALID_ACCOUNT_ID	The application account identifier is invalid
0x80043075	ST_ERROR_INVALID_PRODUCT_ID	The product identifier is invalid
0x80043076	ST_ERROR_INVALID_SERIAL_NUMBER	The product serial number is invalid
0x80043077	ST_ERROR_INVALID_APPID	The application identifier is invalid
0x80043078	ST_ERROR_INVALID_APIKEY	The specified authorization key is invalid
0x80043079	ST_ERROR_ACCOUNT_EXISTS	The application account identifier already exists
0x8004307A	ST_ERROR_ACCOUNT_NOT_CREATED	The application account identifier was not created
0x8004307B	ST_ERROR_ACCOUNT_NOT_FOUND	The application account identifier was not

		found
0x8004307C	ST_ERROR_ACCOUNT_NOT_EXPIRED	Access to this account has not expired
0x8004307D	ST_ERROR_ACCOUNT_NOT_UPDATED	The application account could not be updated
0x8004307E	ST_ERROR_ACCOUNT_EXPIRED	Access to this account has expired
0x8004307F	ST_ERROR_ACCOUNT_REVOKED	Access to this account has been revoked
0x80043080	ST_ERROR_APIKEY_NOT_CREATED	The application key could not be created
0x80043081	ST_ERROR_APIKEY_NOT_FOUND	The application key could not be found
0x80043082	ST_ERROR_APIKEY_NOT_EXPIRED	The application key has not expired
0x80043083	ST_ERROR_APIKEY_NOT_UNIQUE	The application key identifier is not unique
0x80043084	ST_ERROR_APIKEY_NOT_UPDATED	The application key could not be updated
0x80043085	ST_ERROR_APIKEY_NOT_DELETED	The application key could not be deleted
0x80043086	ST_ERROR_APIKEY_EXISTS	The application key already exists
0x80043087	ST_ERROR_APIKEY_EXPIRED	The application key has expired and must be refreshed
0x80043088	ST_ERROR_APIKEY_REVOKED	The application key has been revoked
0x80043089	ST_ERROR_APIKEY_APPID	The application was not found or was not specified
0x8004308A	ST_ERROR_INVALID_TOKEN	The access token is invalid or was not specified
0x8004308B	ST_ERROR_TOKEN_NOT_CREATED	The access token could not be created
0x8004308C	ST_ERROR_TOKEN_NOT_FOUND	The access token could not be found
0x8004308D	ST_ERROR_TOKEN_NOT_EXPIRED	The access token has not expired
0x8004308E	ST_ERROR_TOKEN_NOT_UPDATED	The access token was not updated
0x8004308F	ST_ERROR_TOKEN_NOT_DELETED	The access token could not be deleted
0x80043090	ST_ERROR_TOKEN_EXPIRED	The access token has expired and must be refreshed
0x80043091	ST_ERROR_TOKEN_REVOKED	The access token has been revoked
0x80043092	ST_ERROR_NO_APIKEYS_FOUND	No application keys found for this account
0x80043093	ST_ERROR_NO_TOKENS_FOUND	No access tokens found for this application key
0x80043094	ST_ERROR_NO_TOKENS_REVOKED	No access tokens have been revoked
0x80043095	ST_ERROR_INVALID_STORAGE_OBJECT	Invalid storage object identifier
0x80043096	ST_ERROR_STORAGE_OBJECT_READONLY	The storage object is read-only
0x80043097	ST_ERROR_STORAGE_OBJECT_EXPIRED	Access to the storage object has expired

0x80043098	ST_ERROR_STORAGE_OBJECT_SIZE	The storage object size exceeds storage limits
0x80043099	ST_ERROR_STORAGE_OBJECT_DIGEST	The storage object digest is invalid or cannot be computed
0x8004309A	ST_ERROR_STORAGE_OBJECT_EXISTS	A storage object with this label already exists
0x8004309B	ST_ERROR_STORAGE_OBJECT_MODIFIED	A storage object with this label has been modified
0x8004309C	ST_ERROR_STORAGE_OBJECT_NOT_OWNER	The current user is not the storage object owner
0x8004309D	ST_ERROR_STORAGE_OBJECT_NOT_FOUND	The specified storage object does not exist
0x8004309E	ST_ERROR_STORAGE_OBJECT_NOT_CREATED	The storage object was not created
0x8004309F	ST_ERROR_STORAGE_OBJECT_NOT_MODIFIED	The storage object was not modified
0x800430A0	ST_ERROR_STORAGE_OBJECT_NOT_RENAMED	The storage object was not renamed
0x800430A1	ST_ERROR_STORAGE_FOLDER_EMPTY	The storage folder does not contain any objects
0x800430A2	ST_ERROR_STORAGE_ACCOUNT_QUOTA	The storage account has exceeded its quota
0x800430A3	ST_ERROR_STORAGE_ACCOUNT_LIMIT	The storage account has exceeded its object limit
0x800430A4	ST_ERROR_INVALID_STORAGE_TYPE	The specified storage type is invalid
0x800430A5	ST_ERROR_INVALID_STORAGE_PROVIDER	The specified storage provider is not available
0x800430A6	ST_ERROR_INVALID_STORAGE_REGION	The specified storage region is not available
0x800430A7	ST_ERROR_INVALID_STORAGE_CONTAINER	The storage container does not exist or cannot be accessed
0x800430A8	ST_ERROR_INVALID_STORAGE_LABEL	The storage object label is invalid or undefined
0x800430A9	ST_ERROR_INVALID_QUEUE_HANDLE	The specified queue handle is invalid or the queue has been deleted
0x800430AA	ST_ERROR_INVALID_QUEUE_FILE	The specified file identifier is not valid for this queue
0x800430AB	ST_ERROR_QUEUE_RUNNING	The operation cannot be performed while the queue is running
0x800430AC	ST_ERROR_QUEUE_STOPPED	The operation cannot be performed when the queue has stopped
0x800430AD	ST_ERROR_QUEUE_EMPTY	There are no files in the specified queue
0x800430AE	ST_ERROR_QUEUE_PAUSED	The operation cannot be performed

		while the queue is paused
0x800430AF	ST_ERROR_QUEUE_LOCKED	The operation cannot be performed while the queue is locked
0x800430B0	ST_ERROR_FILE_NOT_QUEUED	The specified file cannot be found in the queue
0x800430B1	ST_ERROR_END_OF_QUEUE	There are no more files in the specified queue
0x800430B2	ST_ERROR_TOO_MANY_FILES	The maximum number of files have been queued for transfer
0x800430B3	ST_ERROR_NO_QUEUED_TRANSFER	No queued file transfer is currently in progress
0x800430B4	ST_ERROR_INVALID_X509_CERTIFICATE	The specified X.509 format certificate is invalid
0x800430B5	ST_ERROR_INVALID_PKCS12_CERTIFICATE	The specified PKCS 12 format certificate is invalid
0x800430B6	ST_ERROR_INVALID_CIPHER_SUITE	The specified cipher suite is invalid or unavailable
0x800430B7	ST_ERROR_DEPRECATED_CIPHER_SUITE	The specified cipher suite is insecure and has been deprecated
0x800430B8	ST_ERROR_INVALID_CERTIFICATE_CHAIN	The certificate chain could not be validated
0x800430B9	ST_ERROR_INVALID_PRIVATE_KEY	The private key for the certificate is invalid
0x800430BA	ST_ERROR_INVALID_API_SESSION	The application session identifier is invalid
0x800430BB	ST_ERROR_EXPIRED_API_SESSION	The application session identifier has expired
0x800430BC	ST_ERROR_INVALID_API_TOKEN	The application token for this session is invalid
0x800430BD	ST_ERROR_EXPIRED_API_TOKEN	The application token for this session has expired
0x800430BE	ST_ERROR_INVALID_API_AUTHID	The authorization token for this session is invalid
0x800430BF	ST_ERROR_INVALID_API_ENDPOINT	The endpoint for the specified request is invalid
0x800430C0	ST_ERROR_INVALID_API_PAYLOAD	The data submitted with the specified request is invalid
0x800430C1	ST_ERROR_UNKNOWN_SESSION_OWNER	The current session owner is unknown or no longer valid
0x800430C2	ST_ERROR_REVOKED_SESSION_AUTH	The authorization token for this session has been revoked
0x800430C3	ST_ERROR_INVALID_URL_SCHEME	The scheme for the specified URL is

		invalid or not supported
0x800430C4	ST_ERROR_INVALID_URL_HOST	The host name for the specified URL is invalid
0x800430C5	ST_ERROR_INVALID_URL_PORT	The port number for the specified URL is invalid
0x800430C6	ST_ERROR_INVALID_URL_PATH	The resource path for the specified URL is invalid
0x800430C7	ST_ERROR_INVALID_CONTENT_TYPE	The content type is invalid or not supported
0x800430C8	ST_ERROR_UNKNOWN_CONTENT_TYPE	The content type cannot be determined
0x800430C9	ST_ERROR_INVALID_CHARSET	The specified character set is invalid or not supported
0x800430CA	ST_ERROR_INVALID_CODEPAGE	The specified code page is invalid or not supported
0x800430CB	ST_ERROR_INVALID_HEADER_TYPE	The specified header type is invalid or not supported
0x800430CC	ST_ERROR_INVALID_HEADER_NAME	The specified header name is invalid or not permitted
0x800430CD	ST_ERROR_INVALID_HEADER_VALUE	The specified header value is invalid or not permitted
0x800430CE	ST_ERROR_HEADER_NOT_FOUND	The specified header value is undefined
0x800430CF	ST_ERROR_INVALID_DIGEST_TYPE	The specified message digest type is invalid
0x800430D0	ST_ERROR_INVALID_DIGEST_VALUE	The specified message digest value is invalid
0x800430D1	ST_ERROR_FILE_DIGEST_MISMATCH	The message digest does not match the specified file
0x800430D2	ST_ERROR_FILE_CONTENT_MISMATCH	The contents of the specified files are not identical
0x800430D3	ST_ERROR_FILE_TIME_MISMATCH	The timestamps for the specified files are not identical
0x800430D4	ST_ERROR_INVALID_TIMESTAMP	The timestamp value is invalid or undefined
0x800430D5	ST_ERROR_INVALID_UNICODE	The string contains invalid or undefined Unicode code points
0x800430D6	ST_ERROR_BUFFER_TOO_LARGE	The specified buffer size exceeds the maximum length allowed
0x800430D7	ST_ERROR_DIRECTORY_NOT_CREATED	Unable to create the specified path or directory
0x800430D8	ST_ERROR_INVALID_LOCALE	The specified locale identifier is invalid

0x800430D9	ST_ERROR_INVALID_LANGUAGE	The specified language name or identifier is invalid
0x800430DA	ST_ERROR_GENERAL_POLICY_VIOLATION	The operation was blocked due to a general policy violation
0x800430DB	ST_ERROR_POLICY_FEATURE_DISABLED	Local policy prevents the use of this feature
0x800430DC	ST_ERROR_POLICY_NO_ZONEINFO	Local policy has disabled security zone information
0x800430DD	ST_ERROR_POLICY_DOMAIN_BLOCKED	Local security policy does not permit access to this domain
0x800430DE	ST_ERROR_POLICY_RESOURCE_BLOCKED	Local security policy does not permit access to this resource
0x800430DF	ST_ERROR_POLICY_ADDRESS_BLOCKED	Local security policy does not permit use of this address
0x800430E0	ST_ERROR_POLICY_CREDENTIALS_RESTRICTED	Local policy does not permit the use of stored credentials
0x800430E1	ST_ERROR_POLICY_FILETYPE_RESTRICTED	Local policy does not permit access to this file
0x800430E2	ST_ERROR_POLICY_TRUST_FAILURE	The operation was prevented by local trust policy restrictions
0x800430E3	ST_ERROR_POLICY_RATE_LIMITED	The request exceeds local security policy connection or request limits
0x800430E4	ST_ERROR_POLICY_ENCRYPTION_REQUIRED	The request requires a secure connection
0x800430E5	ST_ERROR_WEBVIEW2_NOT_INSTALLED	The WebView2 control is not installed on this system
0x800430E6	ST_ERROR_INVALID_WEBVIEW2_VERSION	The WebView2 control installed on this system does not meet minimum version requirements
0x800430E7	ST_ERROR_WEBVIEW2_NOT_ATTACHED	An instance of the WebView2 control has not been attached to a parent window
0x800430E8	ST_ERROR_WEBVIEW2_NOT_FOUND	The WebView2 control window could not be found or has been destroyed
0x800430E9	ST_ERROR_NAVIGATION_RESTRICTED	Navigation has been restricted for this session
0x800430EA	ST_ERROR_SECURITY_STATUS_UNKNOWN	The security status for this session cannot be determined
0x800430EB	ST_ERROR_UNSAFE_URL_SCHEME	Access to this resource is restricted due to an unsafe URL scheme
0x800430EC	ST_ERROR_USER_PROFILE_NOT_CREATED	Unable to create the specified user session profile
0x800430ED	ST_ERROR_USER_PROFILE_LOCKED	The user profile has been locked and

		cannot be shared with another process
0x800430EE	ST_ERROR_RATE_LIMIT_EXCEEDED	Too many requests have been submitted to the server
0x800430EF	ST_ERROR_PROTOCOL_VERSION_REQUIRED	The requested operation requires a different protocol version
0x800430F0	ST_ERROR_INVALID_PROXY_URL	The specified proxy URL is invalid or not a supported proxy type
0x800430F1	ST_ERROR_INVALID_CREDENTIAL_TYPE	The specified credential type is invalid or undefined
0x800430F2	ST_ERROR_INVALID_CREDENTIAL_NAME	The specified credential name is invalid or cannot be found
0x800430F3	ST_ERROR_INVALID_CREDENTIAL_SECRET	The specified credential exists but does not contain a valid secret
0x800430F4	ST_ERROR_INVALID_PROVIDER_ID	The specified service provider ID is invalid or unknown
0x800430F5	ST_ERROR_INVALID_PROVIDER_GUID	The specified service provider GUID is invalid or unknown
0x800430F6	ST_ERROR_PROVIDER_NOT_SUPPORTED	The service provider is not supported in this context
0x800430F7	ST_ERROR_NO_PROVIDER_BASE_URL	A base connection URL must be specified for this service provider
0x800430F8	ST_ERROR_NO_PROVIDER_MODELS	The service provider does not support enumerating models
0x800430F9	ST_ERROR_PROVIDER_FORMAT_CHANGED	The response format returned by the provider was not recognized or is incompatible
0x800430FA	ST_ERROR_MODEL_NAME_REQUIRED	Provider requires a model name, no default model is defined
0x800430FB	ST_ERROR_INVALID_MODEL_NAME	The specified model name is not valid for this provider
0x800430FC	ST_ERROR_INVALID_MODEL_OBJECT	An invalid model object was returned by the provider
0x800430FD	ST_ERROR_MODEL_UNAVAILABLE	The specified model is unavailable for this session
0x800430FE	ST_ERROR_MODEL_NOT_FOUND	The requested model was not found or is not available from the provider
0x800430FF	ST_ERROR_API_VERSION_REQUIRED	An API version has not been specified and it is required by the provider
0x80043100	ST_ERROR_INVALID_API_VERSION	The specified API version is invalid or undefined
0x80043101	ST_ERROR_INVALID_REQUEST_TYPE	The request is not supported by the

		service provider
0x80043102	ST_ERROR_INVALID_PROJECT	The specified project identifier is invalid or unknown
0x80043103	ST_ERROR_SYSTEM_PROMPT_TOO_LONG	The system prompt exceeds the maximum length allowed
0x80043104	ST_ERROR_INVALID_MESSAGE_ID	The specified message identifier is invalid
0x80043105	ST_ERROR_INVALID_MESSAGE_ROLE	The role assigned to this message is invalid
0x80043106	ST_ERROR_TOO_MANY_MESSAGES	The request exceeds the maximum number of messages allowed
0x80043107	ST_ERROR_NO_MESSAGES_FOUND	The are no messages which match the specified criteria
0x80043108	ST_ERROR_END_OF_MESSAGES	There are no more messages available
0x80043109	ST_ERROR_NO_MESSAGE_HISTORY	There are no messages available

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Class Library

Transfer files between a local and server and perform common file management methods on the server.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CFtpClient
File Name	CSFTPV12.DLL
Version	12.0.3024.1286
LibID	A86C8154-EC75-4C39-BB05-B698A87E98F1
Import Library	CSFTPV12.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

The File Transfer Protocol (FTP) library provides a comprehensive API which supports both high level operations, such as uploading or downloading files, as well as a collection of lower-level file I/O functions. In addition to file transfers, an application can create, rename and delete files and directories, search for files using wildcards and perform other common file management functions.

Files can be stored on the local file system or in memory, depending on the needs of your application and multiple file transfers be performed using a single method call. The class can also be used to manage files on the server and supports many of the common protocol extensions that can be used to access the remote file system. It understands a number of different directory listing formats, including those typically used with UNIX and Linux based systems, Windows servers and several other platforms.

This class supports active and passive mode file transfers, firewall compatibility options, proxy servers and secure file transfers using TLS 1.3 and SFTP (SSH 2.0). Secure file transfers support implicit and explicit TLS sessions, client certificates and 256-bit AES encryption. SFTP transfers support public/private key authentication using either RSA or OpenSSH formats.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Class Methods

Class	Description
CFtpClient	Constructor which initializes the current instance of the class.
~CFtpClient	Destructor which releases resources allocated by the class.
Method	Description
Allocate	Allocate the specified number of bytes on the server.
AttachHandle	Attach the specified client handle to this instance of the class.
AttachThread	Attach the specified client handle to another thread.
Cancel	Cancel the current blocking operation.
ChangeDirectory	Change the current working directory on the server.
ChangeDirectoryUp	Change the current working directory on the server.
CloseDirectory	Close the open directory on the server.
CloseFile	Close the current file on the server.
Command	Send a command to the server.
Connect	Establish a client connection with a server.
ConnectUrl	Establish a client connection using the specified URL.
CreateDirectory	Create the specified directory on the server.
CreateSecurityCredentials	Create a new security credentials structure.
DeleteFile	Delete a file from the server.
DeleteSecurityCredentials	Delete a previously created security credentials structure.
DetachHandle	Detach the handle for the current instance of this class.
DisableEvents	Disable event notification.
DisableTrace	Disable logging of network function calls.
Disconnect	Disconnect the client session from the server.
DownloadFile	Download a file from the server to the local system.
EnableEvents	Enable event notification.
EnableFeature	Enable the specified feature in the client.
EnableTrace	Enable logging of network function calls to a file.
FreezeEvents	Suspend and resume event handling by the client.
FtpEventProc	Callback function that processes events generated by the client.
GetActivePorts	Return the range of local port numbers used for active transfers.
GetAutoFileType	Return the file transfer type based on the file extension or content.
GetBufferSize	Return the size of an internal buffer used during data transfers.
GetChannelMode	Return the mode for the specified communication channel.

GetClientQuota	Return quota information for the current client session.
GetData	Copy the contents of a remote file to a local buffer.
GetDefaultZoneId	Returns the default security zone information for downloaded files.
GetDirectory	Get the current working directory on the server.
GetDirectoryFormat	Get the format which is used by the server to list files.
GetErrorString	Return a description for the specified error code.
GetFeatures	Return the features available to the client.
GetFile	Copy a file from the server to the local system.
GetFileList	Return an unparsed list of files in a string buffer.
GetFileNameEncoding	Return the character encoding used when sending a file name to the server.
GetFilePathUrl	Return a normalized URL for the specified file on the server.
GetFilePermissions	Return the access permissions for the specified file.
GetFileSize	Return the size of a file on the server.
GetFileStatus	Return file status information from the server.
GetFileTime	Return the modification time for the specified file on the server.
GetFileType	Return the default file type for the current session.
GetFileZoneInfo	Return file zone identifier information for the specified local file.
GetFirstFile	Return the first file from the file list returned by the server.
GetHandle	Return the client handle used by this instance of the class.
GetIdleTimeout	Return the idle timeout period for the control channel.
GetLastError	Return the last error code.
GetMultipleFiles	Copy multiple files from the server to the local system.
GetNextFile	Return the next file from the file list returned by the server.
GetPriority	Return the current priority for file transfers.
GetProxyType	Return the proxy type selected by the client.
GetResultCode	Return the result code from the previous command.
GetResultString	Return the result string from the previous command.
GetSecurityInformation	Return security information about the current client connection.
GetSecurityZoneId	Returns the security zone identifier for the specified resource.
GetServerInformation	Get system information about the server.
GetServerStatus	Return system status of server.
GetServerTimeZone	Return the timezone offset in seconds for the current server.
GetServerType	Return the type of operating system the server is running on.
GetStatus	Return the current client status.

GetText	Download the contents of a text file to a string buffer.
GetTimeout	Return the number of seconds until an operation times out.
GetTransferStatus	Return file transfer statistics.
IsBlocking	Determine if the current operation is blocked.
IsConnected	Determine if the client is connected to the server.
IsInitialized	Determine if the class has been successfully initialized.
IsReadable	Determine if the client can read data from the data channel.
IsWritable	Determine if the client can write data to the data channel.
Login	Login to the server.
Logout	Logout from the server.
MountStructure	Mount a structure (filesystem) on the server.
NormalizePath	Normalize the specified file name returning a canonical path.
NormalizeUrl	Normalize the specified URL returning a canonical path.
OpenDirectory	Open the specified directory for reading.
OpenFile	Open the specified file for reading or writing.
ProxyConnect	Establish a connection with a proxy server.
PutData	Create a file on the server using the contents of a local buffer.
PutFile	Copy a file from the local system to the server.
PutMultipleFiles	Copy multiple files from the local system to the server.
PutText	Create a text file on the server from the contents of a string buffer.
Read	Read data from the server.
RegisterEvent	Register an event handler for the specified event.
RegisterFileType	Associate a file name extension with a specific file type.
RemoveDirectory	Remove a directory from the server.
RenameFile	Rename a file on the server.
Reset	Reset the client connection.
SetActivePorts	Set the range of local port numbers used for active transfers.
SetBufferSize	Set the size of an internal buffer used during data transfers.
SetChannelMode	Change the security mode for the specified channel.
SetDefaultZoneId	Set the default security zone for downloaded files.
SetDirectoryFormat	Set the format which is used by the server to list files.
SetFeatures	Set the features which can be used by the client.
SetFileMode	Set the current file mode.
SetFileNameEncoding	Set the character encoding type used when sending a file name to the server.
SetFilePermissions	Set the access permissions for the specified file.

SetFileStructure	Set the current file data structure.
SetFileTime	Set the modification time for the specified file on the server.
SetFileType	Set the default file type for the current session.
SetFileZoneInfo	Set or remove the zone indentifier associated with the specified file.
SetIdleTimeout	Set the idle timeout period for the control channel.
SetLastError	Set the last error code.
SetPassiveMode	Set the server in passive mode.
SetPriority	Set the priority for file transfers.
SetTimeout	Set the number of seconds until an operation times out.
ShowError	Display a message box with a description of the specified error.
UploadFile	Upload a file from the local system to the server.
ValidateHostName	Validate the specified host name and return the resolved IP address.
ValidateUrl	Check the contents of a string to ensure it represents a valid URL.
VerifyFile	Compare the contents of a local file against a file stored on the server.
Write	Write data to the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::~CFtpClient

```
~CFtpClient();
```

The **CFtpClient** destructor releases resources allocated by the current instance of the **CFtpClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CFtpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CFtpClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Allocate Method

```
INT Allocate(
   DWORD dwFileLength,
   DWORD dwRecSize
);
```

The **Allocate** method instructs the server to reserve sufficient storage to accommodate the new file being transferred.

Parameters

dwFileLength

The number of bytes to allocate storage for on the server.

dwRecSize

The maximum record or page size for the file. A value of zero indicates that the file does not have a record or page structure, and the parameter is ignored.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be called immediately before the **OpenFile** method.

This method is ignored by those servers which do not require that the maximum size of the file be declared beforehand. The most common FTP servers running under UNIX and Windows do not require that file space be pre-allocated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

OpenFile, SetFileMode, SetFileStructure, SetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::CFtpClient

```
CFtpClient();
```

The **CFtpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **FtpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

The instance of this class is initially attached to the thread which created it and only the owning thread may invoke methods in that instance. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

To enable any thread to invoke class methods regardless of ownership, specify the FTP_OPTION_FREETHREAD option when calling the **Connect** or **ConnectUrl** method. It is important to note this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the instance is synchronized. If one thread invokes a method, it must ensure no other thread will invoke another method at the same time for the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CFtpClient, AttachHandle, AttachThread, Connect, ConnectUrl, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::ChangeDirectory Method

```
INT ChangeDirectory(
   LPCTSTR lpszDirectory
);
```

The **ChangeDirectory** method changes the current working directory for the client session.

Parameters

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the CWD command to change the current working directory. The user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectoryUp, CloseDirectory, Command, GetDirectory, GetFirstFile, GetNextFile, GetResultCode, GetResultString, OpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::ChangeDirectoryUp Method

```
INT ChangeDirectoryUp();
```

The **ChangeDirectoryUp** method changes directory to the parent of the current working directory.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the CDUP command to the server. This command is a special case of the CWD command, and is included to simplify transferring between directory trees on those operating systems which have different syntaxes for naming the parent directory. The current user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, Command, GetDirectory, GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::CloseDirectory Method

```
INT CloseDirectory();
```

The **CloseDirectory** method closes the data socket connection to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method must be called after all of the file information from the server has been returned. Because directory information is returned on the data channel, no file transfers can take place while a directory is being read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetDirectoryFormat, GetFileStatus, GetFirstFile, GetNextFile, OpenDirectory, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::CloseFile Method

```
INT CloseFile();
```

The **CloseFile** method flushes the internal client buffers and closes the data socket connection to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the socket is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the method returns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

OpenFile, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Command Method

```
INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should only be used when the application needs to send a custom, site-specific command or send a command that is not directly supported by this class. This method should never be used to issue a command that opens a data channel. If the application needs to transform data as it is being sent or received, and cannot use the **GetFile** or **PutFile** methods, then use the **OpenFile** method to open a data channel with the server.

By default, file names which are sent to the server using the **Command** method are sent as ANSI characters. If the Unicode version of the function is used, the file name will be converted from Unicode to ANSI using the current codepage. If the server supports UTF-8 encoded file names, the **SetFileNameEncoding** function can be used to specify that file names with non-ASCII characters should be sent as UTF-8 encoded values. It is important to note that this option is only available if the server advertises support for UTF-8 and permits that encoding type.

Because the **Command** method communicates directly with the server using the control connection, the idle timeout period configured with **SetIdleTimeout** applies to this method as well. If a session remains inactive for longer than the configured timeout, the server may close the connection before the command is sent or before a response is received. Adjusting the idle timeout can help prevent unexpected disconnects during periods of infrequent command activity, especially in interactive or long-running operations.

This method is not supported when using SFTP because that protocol does not provide a mechanism for sending arbitrary command strings. Unlike the FTP protocol, which uses a command-response model over a control channel, SFTP uses a structured binary protocol without support for custom commands. If this method is called while connected to an SFTP server, it will fail and return an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetFileNameEncoding, GetIdleTimeout, GetResultCode, GetResultString, OpenFile, PutFile, SetFileNameEncoding, SetIdleTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If the secure port number is specified, an implicit TLS connection will be established by default.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.

FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_NOPWDNUL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier

	versions. This option has no effect on standard or secure connections using SSL.
FTP_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure

	Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only

	supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **SetChannelMode** method.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Example

```
// Connect to a server using a standard (non-secure) connection on
// the default port. The username and password are not encrypted.

bResult = ftpClient.Connect(lpszRemoteHost,
                            FTP_PORT_DEFAULT,
                            lpszUserName,
```

```
                                lpszPassword,
                                FTP_TIMEOUT,
                                FTP_OPTION_DEFAULT);

// Connect to a server using the default port and then initiate a
// secure connection using the AUTH TLS command

bResult = ftpClient.Connect(lpszRemoteHost,
                                FTP_PORT_DEFAULT,
                                lpszUserName,
                                lpszPassword,
                                FTP_TIMEOUT,
                                FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_EXPLICIT);

// Connect to a server on port 990 and immediately initiate a
// secure connection as soon as the connection is established

bResult = ftpClient.Connect(lpszRemoteHost,
                                FTP_PORT_SECURE,
                                lpszUserName,
                                lpszPassword,
                                FTP_TIMEOUT,
                                FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_IMPLICIT);

// Connect to a server on port 22 using the Secure Shell (SFTP)
// protocol to transfer files

bResult = ftpClient.Connect(lpszRemoteHost,
                                FTP_PORT_SSH,
                                lpszUserName,
                                lpszPassword,
                                FTP_TIMEOUT,
                                FTP_OPTION_SECURE_SHELL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, CreateSecurityCredentials, Disconnect, GetSecurityInformation, ProxyConnect, SetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::ConnectUrl Method

```
BOOL ConnectUrl(
   LPCTSTR lpszUrl,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **ConnectUrl** method establishes a connection with the specified server using a URL.

Parameters

lpszUrl

A null-terminated string which specifies the URL for the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note

	that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_NOPWDNUL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.
FTP_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using TLS.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be

	automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is

	established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ConnectUrl** method is a high-level method that uses an FTP URL to establish a connection with a server. Unlike the other connection related methods such as **Connect**, this method does more than simply connect to the server. It will also authenticate the client session, change the current working

directory and set the default file transfer mode. By default, this method will always place the client in passive mode, ensuring the broadest compatibility with most servers. The **ValidateUrl** method can be used to verify that a URL is valid prior to calling this function.

The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the method will attempt to change the current working directory. The paths in an FTP URL are relative to the home directory of the user account and are not absolute paths starting at the root directory on the server. If a file name is also specified in the URL, it will be ignored and only the file path will be used. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The **ConnectUrl** method is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection methods. If you only need to upload or download a file using a URL, then refer to the **UploadFile** and **DownloadFile** methods.

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **ConnectUrl** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Example

```
CFtpClient ftpClient;

if (!ftpClient.ConnectUrl(_T("ftp://ftp.sockettools.com/")))
{
    ftpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect, DownloadFile, UploadFile, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::CreateDirectory Method

```
INT CreateDirectory(
   LPCTSTR lpszDirectory
);
```

The **CreateDirectory** method creates the specified directory on the server.

Parameters

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the MKD command to create the directory. The user must have the appropriate permission to create the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, GetDirectory, RemoveDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

A value which specifies one or more options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the

current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The

dwOptions parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              FTP_PORT_SECURE,
                              FTP_TIMEOUT,
                              FTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::DeleteFile Method

```
INT DeleteFile(
   LPCTSTR lpszFileName
);
```

The **DeleteFile** method deletes the specified file from the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The current user must have the appropriate permission to delete the file, or an error will be returned by the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, PutFile, RenameFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Connect, ProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::DownloadFile Method

```
BOOL DownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPFTPTRANSFERSTATUS lpStatus
   FTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **DownloadFile** method downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the complete URL of the file to be downloaded to the local system. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be

	necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DownloadFile** method provides a convenient way for an application to download a file in a single method call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then download the file to the local system. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the method will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be downloaded as a text or binary file. A value of "a" specifies that the file should be downloaded as a text file. A value of "i" specifies that the file should be downloaded as a binary file. If the type is not explicitly specified, the

file will be downloaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **DownloadFile** method is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **GetFile** to download the file.

Example

```
CFtpClient ftpClient;
CString strLocalFile = _T("c:\\temp\\database.mdb");
CString strFileURL = _T("ftp://ftp.example.com/updates/database.mdb");

// Download the file using the specified URL
if (!ftpClient.DownloadFile(strLocalFile, strFileURL))
{
    ftpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, GetFile, GetFileZoneInfo, GetTransferStatus, SetFileZoneInfo, UploadFile, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages

	will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event occurs when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event occurs when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::EnableFeature Method

```
BOOL EnableFeature(
   DWORD dwFeature,
   BOOL bEnable
);
```

The **EnableFeature** method enables or disables a specific server feature available to the client.

Parameters

dwFeature

An unsigned integer which specifies the feature to be enabled to disabled for the current client session. Refer to the documentation for the **GetFeatures** method for a list of available features.

bEnabled

A boolean flag which specifies if the feature should be enabled or disabled. If the value is non-zero, the library will attempt to use that feature on the server. If the value is zero, the feature is disabled. If an application calls a method which requires a specific feature and that feature is disabled, the method will fail with an error indicating the feature is not supported.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero. To get extended error information, call the **GetLastError** method.

Remarks

The **EnableFeature** method is used to enable or disable a specific feature for the current session. When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. An application can use the **EnableFeature** method to control what commands will be sent to the server, or re-enable a command that was previously disabled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetFeatures, SetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::EnumFiles Method

```
INT EnumFiles(
   LPCTSTR lpszDirectory,
   LPCTSTR lpszFileMask,
   DWORD dwOptions,
   LPFTPFILESTATUSEX lpFileList,
   INT nMaxFiles
);
```

The **EnumFiles** method populates an array of structures that contain information about the files in a directory.

Parameters

lpszDirectory

A null-terminated string that specifies the name of a directory on the server. If this parameter is NULL or points to an empty string, the method will return the files in the current working directory. This string cannot contain wildcard characters and must specify a valid directory name that exists on the server.

lpszFileMask

A null-terminated string that specifies a wildcard file mask that is used to return a subset of files in the directory. If this parameter is NULL or an empty string then all of the files in the directory will be returned.

dwOptions

An unsigned integer value that specifies one or more options. This parameter can be a combination of one or more of the following values:

Value	Description
FTP_ENUM_DEFAULT	The method will return both regular files and subdirectories.
FTP_ENUM_FILE	The method will return only regular files.
FTP_ENUM_DIRECTORY	The method will return only subdirectories.
FTP_ENUM_FULLPATH	The method will return the full path of the file or subdirectory.

lpFileList

A pointer to an array of FTPFILESTATUSEX structures which contains information about each of the files in the specified directory. This parameter cannot be NULL, and the array must be large enough to store the number of files specified by the ***nMaxFiles*** parameter.

nMaxFiles

An integer value that specifies the maximum number of files that should be returned. This value must be greater than zero and the ***lpFileList*** parameter must provide an array that is large enough to store information about each file.

Return Value

If the method succeeds, the return value is the number of files returned by the method. If the directory is empty or there are no files that match the specified wildcard file mask, the method will return zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnumFiles** method provides a high-level interface for obtaining a list of available files in a directory on the server in a single function call. This is an alternative to opening a directory and returning information about each file by calling the **GetNextFile** method in a loop.

This method temporarily changes the current working directory to the directory specified by the *lpszDirectory* parameter. The current working directory will be restored to its original value when the method returns. The user must have the appropriate permissions to access the directory or this method will fail.

To obtain information on a subset of files in the directory, you can specify a wildcard file mask. For FTP and FTPS (SSL) sessions, this value is passed as a parameter to the LIST command and the server performs the wildcard matching. For SFTP (SSH) sessions the wildcard matching is performed by the library, and the standard conventions for Windows file wildcards are used.

By default, the *szFileName* member for each **FTPFILESTATUSEX** structure will contain the base file name. If the FTP_ENUM_FULLPATH option is specified, the method will return the full path name to the file. The library must be able to automatically determine the path delimiter that is used by the server. This is done by examining how the server identifies itself, the current directory format and the path the server returns for the current working directory. For example, UNIX based servers use the forward slash as a path delimiter. If the method cannot determine what the appropriate path delimiter is, it will ignore this option and return only the base file name.

This method will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
LPFTPFILESTATUSEX lpFileList = new FTPFILESTATUSEX[MAXFILECOUNT];

// Return all of the regular files in the current working directory
INT nResult = pClient->EnumFiles(FTP_ENUM_FILE, lpFileList, MAXFILECOUNT);

if (nResult == FTP_ERROR)
{
    DWORD dwError = pClient->GetLastError();
    _tprintf(_T("EnumFiles failed, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("EnumFiles returned %d files\n"), nResult);
for (INT nIndex = 0; nIndex < nResult; nIndex++)
{
    _tprintf(_T("file=\"%s\" size=%I64d\n"), lpFileList[nIndex].szFileName,
                                             lpFileList[nIndex].uiFileSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileStatus, GetFirstFile, GetNextFile, OpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetActivePorts Method

```
INT GetActivePorts(
   UINT *lpnLowPort,
   UINT *lpnHighPort
);
```

The **GetActivePorts** method returns the local port numbers used for active mode file transfers.

Parameters

lpnLowPort

Points to an unsigned integer that will contain the low port number when the function returns.

lpnHighPort

Points to an unsigned integer that will contain the high port number when the function returns.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method is used to determine the default port numbers being used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **SetActivePorts** method can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetActivePorts, SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetAutoFileType Method

```
UINT GetAutoFileType(
   LPCTSTR lpszFileName,
   BOOL bScanFile
);
```

The **GetAutoFileType** method returns the file transfer type based on the file extension or content.

Parameters

lpszFileName

A null-terminated string which specifies the local path to the file. This parameter cannot be NULL and cannot specify a local device name or directory.

bScanFile

An optional value which specifies if the contents of the file should be scanned. A value of zero indicates that only the file extension should be used to determine the file type, while a non-zero value specifies the contents of the file should be examined if the file type cannot be determined based on its extension. This parameter value is zero (false) by default.

Return Value

If the method succeeds, the return value is the file transfer type. If the method fails, the return value is INVALID_FILE_TYPE. To get extended error information, call **GetLastError**.

Remarks

This method is used to determine the file transfer type to be used when uploading or downloading files. This method is called internally when FILE_TYPE_AUTO is specified as the default file type. The return value may be one of the following:

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.

If the file extension or contents are not recognized, the default file transfer type for the client session will be returned. This will usually be FILE_TYPE_IMAGE, however this can be changed by calling the **RegisterFileType** method. The file type for the current client session can be explicitly set using the

SetFileType method.

If the ***bScanFile*** parameter is non-zero, the local file will be opened in a shared reading mode and up to 4,096 bytes will be examined to determine if it contains binary data. If the file is currently locked or has been opened exclusively by another process, the file type associated with the file extension will be returned instead. Text files which contain UTF-16 text will always return a file type of FILE_TYPE_IMAGE because they can contain non-ASCII characters and/or embedded null characters.

If the ***bScanFile*** parameter is non-zero and the file type cannot be determined based on the file name extension, the file specified by ***lpszFileName*** must exist and be a regular file. If the file does not exist, an error will be returned and the last error code will be set to ST_ERROR_FILE_NOT_FOUND. If the ***bScanFile*** parameter is zero, no errors will be returned if the file does not exist, the function will only check the file name extension to determine the file type. When downloading a file, the ***bScanFile*** parameter should normally be zero because the local file may not exist yet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileType, RegisterFileType, SetFileMode, SetFileStructure, SetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetBufferSize Method

```
INT GetBufferSize();
```

The **GetBufferSize** method returns the size in bytes of an internal buffer that will be used during data transfers.

Parameters

None.

Return Value

If the method succeeds, the return value is the size of the internal buffer that will be used in data transfers. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a value of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

SetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetChannelMode Method

```
INT GetChannelMode(
   INT nChannel
);
```

The **GetChannelMode** method returns the mode of the specified communications channel.

Parameters

nChannel

An integer value which specifies which channel to return information for. It may be one of the following values:

Value	Description
FTP_CHANNEL_COMMAND	Return information about the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Return information about the data channel. This is the communication channel used to send or receive data during a file transfer.

Return Value

If the method succeeds, the return value is the mode for the specified channel. If the method fails, it will return FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetChannelMode** method returns a integer which may be one of the following values:

Value	Description
FTP_CHANNEL_CLEAR	The channel is not encrypted. This is the default mode for both channels when a standard, non-secure connection is established with the server.
FTP_CHANNEL_SECURE	The channel is encrypted. This is the default mode for both channels when a secure connection is established with the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

SetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetClientQuota Method

```
INT GetClientQuota(
   LPFTPCLIENTQUOTA lpClientQuota
);
```

The **GetClientQuota** method returns information about file quotas for the current client session.

Parameters

lpClientQuota

A pointer to an FTPCLIENTQUOTA structure which contains the quota information returned by the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the XQUOTA command to obtain information for the current client session. If the server does not support this command, the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetServerInformation, GetServerType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetData Method

```
INT GetData(
   LPCTSTR lpszRemoteFile,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

INT GetData(
   LPCTSTR lpszRemoteFile,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

INT GetData(
   LPCTSTR lpszRemoteFile,
   LPTSTR lpBuffer,
   DWORD dwMaxLength
);

INT GetData(
   LPCTSTR lpszRemoteFile,
   CString& strBuffer
);
```

The **GetData** method transfers the contents of a file on the server to the specified buffer.

Parameters

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

lpBuffer

A buffer which will contain the data transferred from the server. In alternate forms of the method, this argument may also be a pointer to a global memory handle or reference a **CString** object.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpBuffer*** parameter. If the ***lpBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

dwMaxLength

An unsigned integer value which specifies the maximum number of characters can be copied into the ***lpBuffer*** string. This parameter is used with the version of the method that returns the data in a character array and includes the terminating null character. This value must be greater than one and the ***lpBuffer*** parameter cannot be NULL, otherwise the method will return an error.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetData** method is used to download the contents of a remote file into a local buffer. The

method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the file data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the third method or fourth method is used, where the data is returned in a string buffer, the data may be modified so that the end-of-line character sequence matches the convention used by the Windows platform (a carriage return character followed by a linefeed). If Unicode is being used, the data will be converted from a byte array to a Unicode string. An application should only use these versions of the **GetData** method if the remote file contains text.

It is important for your application to initialize the value of the *lpdwLength* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when downloading the file, the variable passed to this method may have a value of zero when the method returns, indicating no data has been returned to the caller.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = pClient->GetData(lpszRemoteFile, &hgblBuffer, &cbBuffer);

if (nResult != FTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, EnableEvents, GetFile, GetTransferStatus, PutData, PutFile, RegisterEvent, SetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetDefaultZoneId Method

```
BOOL GetDefaultZoneId(
   LPDWORD lpdwZoneId
);
```

The **GetDefaultZoneId** method returns the default security zone for downloaded files.

Parameters

lpdwZoneId

A pointer to an unsigned integer which is updated with the default zone ID for the client session. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the method succeeds, the return value is non-zero and *lpdwZoneId* will be updated with the default zone identifier. If no default zone has been explicitly set, the function will still return success and *lpdwZoneId* will be set to `SECURITY_ZONE_NONE`. If the method fails, it returns zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetDefaultZoneId** method returns the zone identifier which was previously set using the **SetDefaultZoneId** method. If no default zone has been specified for the client session, the default behavior is to not set a security zone for the file. This maintains backwards compatibility with previous versions of the library which would not create the security zone metadata for local files.

This method is typically used to confirm the currently configured zone before downloading a file, or to verify whether the default behavior has been overridden. The default zone ID is specific to the client session and does not affect other sessions or system-wide settings.

To get the security zone identifier for a local file, use the **GetFileZoneInfo** method. To change or remove the security zone information for a file, use the **SetFileZoneInfo** method.

For more detailed information about the zone identifiers, refer to the **SetDefaultZoneId** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetFileZoneInfo, SetDefaultZoneId, SetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetDirectory Method

```
INT GetDirectory(
   LPTSTR lpszDirectory,
   INT nMaxLength
);
```

The **GetDirectory** method copies the current working directory on the server to the specified buffer.

Parameters

lpszDirectory

Points to a buffer that will contain the name of the current working directory on the server. The file pathing and name conventions must be that of the server.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null-character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the PWD command to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, CreateDirectory, RemoveDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetDirectoryFormat Method

```
INT GetDirectoryFormat();
```

The **GetDirectoryFormat** method returns an identifier which specifies what format is being used by the server to list files. By default, the library will automatically determine the appropriate format, but this value may be overridden by the **SetDirectoryFormat** method.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Refer to the **SetDirectoryFormat** method for a list of directory format types that are supported by the library. This method can be used to determine which format was selected by the library after a file listing has been retrieved.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CloseDirectory, GetFileStatus, GetFirstFile, GetNextFile, OpenDirectory, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFeatures Method

```
DWORD GetFeatures();
```

The **GetFeatures** method returns the server features available to the client.

Parameters

None.

Return Value

If the method succeeds, the return value is one or more bit flags which specify the features that are available to the client. If the method fails, it will return zero. Because it is possible that no features would be enabled, a return value of zero does not always indicate an error. An application should call **GetLastError** to determine if an error code has been set.

Remarks

The **GetFeatures** method returns a value which may be a combination of one or more of the following bit flags:

Value	Description
FTP_FEATURE_SIZE	The server supports the SIZE command to determine the size of a file. If this feature is not enabled, the library will attempt to use the MLST or STAT command to determine the file size.
FTP_FEATURE_STAT	The server supports using the STAT command to return information about a specific file. If this feature is not enabled, the client may not be able to obtain information about a specific file such as its size, permissions or modification time.
FTP_FEATURE_MDTM	The server supports the MDTM command to obtain information about the modification time for a specific file. This command may also be used to set the file time on the server.
FTP_FEATURE_REST	The server supports restarting file transfers using the REST command. If this feature is not enabled, the client will not be able to restart file transfers and must upload or download the complete file.
FTP_FEATURE_SITE	The server supports site specific commands using the SITE command. If this feature is not enabled, no site specific commands will be sent to the server.
FTP_FEATURE_IDLE	The server supports setting the idle timeout period using the SITE IDLE command to specify the number of seconds that the client may idle before the server terminates the connection.
FTP_FEATURE_CHMOD	The server supports modifying the permissions of a specific file using the SITE CHMOD command. If this feature is not enabled, the client will not be able to set the permissions for a file.
FTP_FEATURE_AUTH	The server supports explicit TLS sessions using the AUTH command. If this feature is not enabled, the client will only be able to connect to a secure server that uses implicit TLS connections. Changing this feature has no effect on standard,

	non-secure connections.
FTP_FEATURE_PBSZ	The server supports the PBSZ command which specifies the buffer size used with secure data connections. If this feature is disabled, it may prevent the client from changing the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PROT	The server supports the PROT command which specifies the protection level for the data channel. If this feature is disabled, the client will be unable to change the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_CCC	The server supports the CCC command which returns the command channel to a non-secure mode. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_HOST	The server supports the HOST command which enables a client to specify the hostname after establishing a connection with a server that supports virtual hosting.
FTP_FEATURE_MLST	The server supports the MLST command which returns status information for files. If this feature is enabled, the MLST command will be used instead of the STAT command.
FTP_FEATURE_MFMT	The server supports the MFMT command which is used to change the last modification time for a file. If this command is supported, it is used instead of the MDTM command to change the modification time for a file.
FTP_FEATURE_XCRC	The server supports the XCRC command which returns the CRC-32 checksum for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_XMD5	The server supports the XMD5 command which returns an MD5 hash for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_LANG	The server supports the LANG command which sets the language used for the current client session. Command responses and file naming conventions will use the specified language.
FTP_FEATURE_UTF8	The server supports the OPTS UTF-8 command which specifies UTF-8 encoding when specifying filenames. This feature is typically used in conjunction with setting the default language for the client session.
FTP_FEATURE_XQUOTA	The server supports the XQUOTA command which returns quota information for the current client session.
FTP_FEATURE_UTIME	The server supports the UTIME command which is used to change the last modification time for a specified file.

When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the

server reports that the command is unrecognized that feature will automatically be disabled in the client.

For example, the first time an application calls the **GetFileSize** method to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

EnableFeature, SetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFile Method

```
INT GetFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
   DWORD dwOffset
);
```

The **GetFile** method transfers the specified file on the server to the local system.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be downloaded to the local system. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, EnableEvents, GetMultipleFiles, GetTransferStatus, PutFile, PutMultipleFiles, RegisterEvent, SetBufferSize, VerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileList Method

```
INT GetFileList(
   LPCTSTR lpszDirectory,
   DWORD dwOptions,
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT GetFileList(
   LPCTSTR lpszDirectory,
   DWORD dwOptions,
   CString& strBuffer
);
```

The **GetFileList** method returns an unparsed list of files in the specified directory.

Parameters

lpszDirectory

A null-terminated string that specifies the name of a directory and/or a wildcard file mask. The format of the directory name must match the file naming conventions of the server. If this parameter is NULL or points to an empty string, the current working directory will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_LIST_DEFAULT	This option specifies the server should return a complete listing of files in the specified directory with as much detail as possible. This typically means that the file size, date, ownership and access rights will be returned to the client. Information about the files are returned in lines of text, with each line terminated by carriage return and linefeed (CRLF) characters. The exact format of the data returned is specific to the server operating system.
FTP_LIST_NAMEONLY	This option specifies the server should only return a list of file names, with no additional information about the file. Each file name is terminated by carriage return and linefeed (CRLF) characters.

lpszBuffer

A null-terminated string buffer that will contain the list of files when the function returns. This buffer should be large enough to store the complete file listing and a terminating null character. If the buffer is smaller than the total amount of data returned by the server, the data will be truncated. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the number of bytes copied into the string buffer, not

including the terminating null character. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **GetFileList** method returns a list of files in the specified directory, copying the data to a string buffer. Unlike the other methods like **EnumFiles** that parse a directory listing and return information in an **FTPFILESTATUS** structure, this method returns the unparsed file list data. The actual format of the data that is returned depends on the operating system and how the server implements file listings. For example, UNIX servers typically return the output from the **/bin/ls** command.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **ChangeDirectory** method and then call this method again, passing NULL or an empty string as the *lpszDirectory* parameter.

This method can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the library. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the FTP_LIST_NAMEONLY option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This method is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
CString strFileList;

nResult = pClient->GetFileList(NULL, FTP_LIST_DEFAULT, strFileList);

if (nResult != FTP_ERROR)
    pEditCtl->SetWindowText(strFileList);
else
{
    pClient->ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseDirectory, EnumFiles, GetDirectoryFormat, GetFirstFile, OpenDirectory, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileNameEncoding Method

```
INT GetFileNameEncoding();
```

The **GetFileNameEncoding** method returns an identifier which specifies what type of encoding is being used when file names are sent to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the type of encoding that is used. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Refer to the **SetFileNameEncoding** method for a list of encoding types that are supported by the library.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Command, EnableFeature GetFeatures, SetFileNameEncoding, SetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFilePathUrl Method

```
INT GetFilePathUrl(
   LPCTSTR lpszFilePath,
   LPTSTR lpszFileUrl,
   INT nMaxLength
);

INT GetFilePathUrl(
   LPCTSTR lpszFilePath,
   CString& strFileUrl
);
```

The **FtpGetFilePathUrl** method returns a normalized URL for the specified file on the server.

Parameters

lpszFilePath

A null-terminated string which specifies the file path on the server. The path may be absolute or relative, and the path does not have to exist on the server. This parameter cannot be NULL.

lpszFileUrl

A null-terminated string buffer that will contain the URL for the specified file. If this parameter is NULL, nothing will be copied but the method will return the length of the URL, not including the terminating null character. An alternate version of this method accepts a reference to a **CString** object.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. This must include space for the terminating null character. If ***lpszFileUrl*** is NULL this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the ***lpszFileUrl*** buffer, not including the terminating null character. If a buffer is provided, it must be large enough to contain the entire URL or the method will fail. If the ***lpszFileUrl*** parameter is NULL, the method will return the length of the URL and nothing will be copied. If the method fails, it will return zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetFilePathUrl** method will return the URL associated with the specified file path. This can be useful if you wish to display or store URLs rather than relative paths to a file on the current server. For example, if you are connected to the server `example.com` on port 21 using a standard connection and you provide a relative file path such as `documents/invoice.pdf` this method would return a URL which looks like `ftp://example.com/documents/invoice.pdf`. For security reasons, the resulting URL will never contain user credentials.

The file path in the URL will be normalized to remove extraneous directory traversals and path separators. If the path is relative, it will be expanded to an absolute path based on the client's current working directory. If an absolute path is provided, it will be included in the URL as-is. It is not required for the file path to exist on the server at the time this method is called. For more information about path normalization, refer to the **NormalizePath** method.

This method can also be used to obtain a URL which can be passed to the **GetSercurityZoneId**

method to determine which security zone should be assigned to the file when it is downloaded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, GetSecurityZoneId, NormalizePath, NormalizeUrl, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFilePermissions Method

```
INT GetFilePermissions(
   LPCTSTR lpszFileName,
   LPDWORD lpdwPermissions
);
```

The **GetFilePermissions** method returns information about the access permissions for a specific file on the server.

Parameters

lpszFileName

A null-terminated string which contains the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

lpdwPermissions

A pointer to an unsigned integer which will contain the access permissions for the file when the method returns. The file permissions are represented as bit flags, and may be one or more of the following values:

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.

FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the method succeeds, the return value is a result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the STAT command to retrieve information about the specified file. On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the method will fail and value pointed to by the ***lpdwPermissions*** parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileStatus, SetFilePermissions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileSize Method

```
INT GetFileSize(
   LPCTSTR lpszFileName,
   LPDWORD lpFileSize
);

INT GetFileSize(
   LPCTSTR lpszFileName,
   ULARGE_INTEGER *lpFileSize
);
```

The **GetFileSize** method returns the size of the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes. For files larger than 4GiB, this parameter should point to an unsigned 64-bit integer.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the method call will fail, and the ***lpFileSize*** parameter will be set to zero.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Some FTP servers will refuse to return the size of a file if the current file type is set to FILE_TYPE_ASCII because the size of a text file on the server may not accurately reflect what the size of the file will be on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileStatus, OpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileStatus Method

```
BOOL GetFileStatus(
   LPCTSTR lpszFileName,
   LPFTPFILESTATUS lpFileStatus
);
```

The **GetFileStatus** method returns information about a specific file on the server.

Parameters

lpszFileName

A null-terminated string which contains the name of the file that status information will be returned on. The file name cannot contain any wildcard characters.

lpFileStatus

A pointer to an FTPFILESTATUS structure which contains information about the file returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method uses the STAT command to retrieve information about the specified file. Unlike the **GetFirstFile** and **GetNextFile** methods, which read through a file list returned on the data channel, this method reads the result of a command string. For applications that need information about a specific file, using this method can be considerably faster than iterating through all of the files in a given directory. Note that not all servers support using the command in this way.

On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the FTPFILESTATUS structure members will be empty strings and zero values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseDirectory, GetDirectoryFormat, GetFirstFile, GetNextFile, GetTransferStatus, OpenDirectory, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileTime Method

```
INT GetFileTime(
   LPCTSTR lpszFileName,
   LPSYSTEMTIME lpFileTime,
   BOOL bLocalize
);
```

The **GetFileTime** method returns the modification time for the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a SYSTEMTIME structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetFileTime** method can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time. If you plan on changing the values returned in the SYSTEMTIME structure and then calling **SetFileTime** method to modify the file time on the server, you should do not localize the time.

This method uses the MDTM command to determine the modification time of the specified file. If the server does not support this command, the method will attempt to use the STAT command to determine the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileStatus, OpenDirectory, SetFileTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileType Method

```
UINT GetFileType();
```

The **GetFileType** method returns the default file type for the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value is an integer value that identifies the current file type. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetFileType** method will return one of the following values:

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **RegisterFileType** method.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executables. If the type of file cannot be automatically determined, it will always be considered a binary file.

If this method is called when connected to an SFTP (SSH) server, the default file type will always be FILE_TYPE_IMAGE because SFTP does not differentiate between text files and binary files.

The **SetFileType** function can be used to change the default file type. To determine the automatic file type for a specific file based on its extension or contents, use the **GetAutoFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetAutoFileType, OpenFile, RegisterFileType, SetFileMode, SetFileStructure, SetFileType,

SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFileZoneInfo Method

```
BOOL GetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   LPDWORD lpdwZoneId,
   LPTSTR lpszHostUrl,
   INT nMaxLength
);

BOOL GetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   LPDWORD lpdwZoneId,
   CString& strHostUrl
);
```

The **GetFileZoneInfo** method returns the security zone information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and read from the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

lpdwZoneId

A pointer to an unsigned integer which will contain the zone ID assigned to the file when the method returns. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string buffer which will contain the URL used to originally download the file. This information is not always available, in which case an empty string will be returned. If this information is not required, this parameter may be NULL and the ***nMaxLength*** parameter should be specified with a value of zero. An alternate version of this method accepts a reference to a **CString** object which will contain the URL when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the ***lpszHostUrl*** string including the terminating null character. If the ***lpszHostUrl*** parameter is NULL,

this value should be zero. If the buffer is not large enough to contain the entire URL, the string will be truncated. The maximum allowed URL length is 2,048 characters.

Return Value

If the method succeeds, the return value is non-zero. If the method fails or zone information is not available, the method returns zero. To get extended error information, call the **GetLastError** method.

Remarks

When a file is downloaded from the Internet or other external sources, Windows may assign a zone identifier to the file using an alternate data stream named `Zone.Identifier`. This metadata helps determine how the system treats the file with respect to security, such as whether to prompt the user with a warning before opening it.

The **GetFileZoneInfo** method retrieves the zone ID associated with the specified local file, allowing an application to determine the origin and trust level of that file. This can be useful when implementing additional security measures, auditing downloaded content, or providing user warnings about potentially untrusted files.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be read from files on network volumes because the SMB protocol does not support alternate data streams.

In addition to the zone ID, this method can optionally return the source URL used to download the file. This information, if present, is extracted from the same alternate data stream. Not all files will include this URL, as it depends on the program or browser that downloaded the file and whether it preserved this metadata. This method will not return an error if the URL stored with the security zone information is not a valid FTP URL. For example, it's possible for this method to return zone information with an HTTP URL. You can use the **ValidateUrl** method to determine if the host URL is a valid FTP URL.

Note that not all file systems support alternate data streams. For example, FAT32 volumes do not support them, and in such cases, zone information will not be available. Additionally, some applications may remove this metadata after download or strip it during file copy operations. The absence of zone information does not guarantee that a file is safe or originated locally.

If you wish to distinguish between a file which exists without any zone information, and an error condition where the file does not exist or could not be accessed, check the last error code. If this method returns zero and the **GetLastError** method returns NO_ERROR, this indicates the file name is valid and it exists, but the `Zone.Identifier` alternate data stream does not exist.

For more detailed information about the security zone identifiers, refer to the **SetDefaultZoneId** method. To change or remove the security zone metadata from a local file, use the **SetFileZoneInfo** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultZoneId, GetFile, GetMultipleFiles, SetDefaultZoneId, SetFileZoneInfo, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetFirstFile Method

```
BOOL GetFirstFile(
   LPFTPFILESTATUS lpFileStatus
);
```

The **GetFirstFile** method returns the first file in the directory listing returned by the server after a call to the **OpenDirectory** method.

Parameters

lpFileStatus

A pointer to an FTPFILESTATUS structure which contains information about the file returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This file list information returned by the server is cached by the library, allowing you to use this method to reposition back to the beginning of the file list.

Example

```
if (pClient->OpenDirectory() != FTP_ERROR)
{
    FTPFILESTATUS ftpFile;
    BOOL bResult;

    bResult = pClient->GetFirstFile(&ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = pClient->GetNextFile(&ftpFile);
    }

    pClient->CloseDirectory();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseDirectory, GetDirectoryFormat, GetFileStatus, GetNextFile, OpenDirectory, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetIdleTimeout Method

```
INT GetIdleTimeout();
```

The **GetIdleTimeout** method returns the idle timeout period for the control channel.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is FTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **GetIdleTimeout** method returns the server-side idle timeout value that was most recently set for the control connection using the **SetIdleTimeout** method. This value represents how long the server will wait during periods of inactivity before automatically closing the connection. If the idle timeout has not been explicitly set during the session, the method will return the default timeout value of 300 seconds (5 minutes).

The timeout value is only available if the server supports the non-standard `SITE IDLE` command. If the server does not support this command, this method will fail. In that case, the default timeout enforced by the server will apply, but the timeout period cannot be accessed or changed by the client. This method will also return an error if the session is using SFTP, as that protocol does not separate control and data channels and does not support idle timeout settings in the same way.

This method does not query the server for the current idle timeout value for the control channel. It simply returns the last known value that was successfully applied during the current session, if available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Command, GetTimeout, SetIdleTimeout, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetMultipleFiles Method

```
INT GetMultipleFiles(
   LPCTSTR lpszLocalDirectory,
   LPCTSTR lpszRemoteDirectory,
   LPCTSTR lpszFileMask
);
```

The **GetMultipleFiles** method copies one or more files from the server to the local host, using the specified wildcard.

Parameters

lpszLocalDirectory

Pointer to a null-terminated string which specifies the local directory where the files will be downloaded to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs. A NULL pointer or empty string specifies that files should be copied to the current working directory.

lpszRemoteDirectory

Pointer to a null-terminated string which specifies the remote directory where the files will be downloaded from. A NULL pointer or empty string specifies that the files should be copied from the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the server to the local system. The file mask should follow the native conventions used for wildcard file matches on the server.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMultipleFiles** method is used to transfer files from the server to the local host which match a specified wildcard file mask. This method requires that the client be able to automatically list and parse directory listings from the server, otherwise an error will be returned. All files will be transferred using the current file type as specified by the **SetFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, GetFile, PutFile, PutMultipleFiles, SetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetNextFile Method

```
BOOL GetNextFile(
   LPFTPFILESTATUS lpFileStatus
);
```

The **GetNextFile** method returns the next file in the directory listing returned by the server.

Parameters

lpFileStatus

A pointer to an FTPFILESTATUS structure which contains information about the file returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextFile** method returns the next file in the directory listing. If the last file has been returned, the method will return zero and the client should call the **CloseDirectory** method to close the directory.

Example

```
if (pClient->OpenDirectory() != FTP_ERROR)
{
    FTPFILESTATUS ftpFile;
    BOOL bResult;

    bResult = pClient->GetFirstFile(&ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = pClient->GetNextFile(&ftpFile);
    }

    pClient->CloseDirectory();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseDirectory, GetDirectoryFormat, GetFileStatus, GetFirstFile, OpenDirectory, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetPriority Method

```
INT GetPriority();
```

The **GetPriority** method returns a value which specifies the priority of file transfers.

Parameters

None.

Return Value

If the method succeeds, the return value is the current file transfer priority. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Value	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetProxyType Method

```
INT GetProxyType();
```

The **GetProxyType** method returns the type of proxy that the client is connected to. By default, no proxy server is specified and this method returns a value of FTP_PROXY_NONE. For a list of possible proxy server types, refer to the **ProxyConnect** method.

Parameters

None.

Return Value

If the method succeeds, the return value identifies the type of proxy that the client is connected to. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, ProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Command, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetResultString Method

```
INT GetResultString(
   LPTSTR lpszResult,
   INT cbResult
);

INT GetResultString(
   CString& strResult
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

The buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetSecurityZoneId Method

```
BOOL GetSecurityZoneId(
   LPCTSTR lpszResource,
   LPDWORD lpdwZoneId
);
```

The **GetSecurityZoneId** method returns the security zone identifier for the specified resource.

Parameters

lpszResource

A null-terminated string which specifies the resource URL to check. This must be either a valid FTP URL or a local file URL.

lpdwZoneId

A pointer to an unsigned integer which is updated with the zone ID for the specified resource. It may be one of the following values when the method returns:

Value	Description
SECURITY_ZONE_NONE	The security zone for this resource cannot be determined.
SECURITY_ZONE_LOCAL	The resource is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The resource originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The resource originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The resource originates from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The resource originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the method succeeds, the return value is non-zero and *lpdwZoneId* will be updated with the zone identifier. If the security zone could not be determined, *lpdwZoneId* will be set to `SECURITY_ZONE_NONE`. If the method fails, it returns zero and the error code can be retrieved by calling the **GetLastError** method.

Remarks

This **GetSecurityZoneId** method is used to determine how a resource should be handled in terms of local security policy before performing a file transfer or download. For example, you may want to determine whether a resource originates from the Internet or local intranet before deciding whether to attach zone metadata, prompt the user, or apply additional security restrictions. It can also be used to implement policy-based rules for accepting or rejecting downloads based on their origin.

The *lpszResource* parameter must use the `ftp://`, `ftps://`, `ftpes://`, `sftp://` or `file://` schemes. This method will not accept URLs for other schemes such as `http://` or `https://`. While this method does not verify that the resource exists, it does require the URL to be syntatically valid. If the host name cannot be resolved to an IPv4 or IPv6 address, the function will fail. If the resource is a

local file, it must be a regular file and cannot be the name of a folder or device.

If the resource is specified using a `file://` URL that refers to a file on a network share (e.g.: `file://server/share/file.txt`), the method will attempt to access that file across the network. This may cause the application to block while Windows resolves the hostname and establishes a connection to the shared volume. If the network share is unavailable or slow to respond, the method may block until the connection succeeds or the attempt times out.

If this method returns FALSE, this indicates either the resource type is not supported or there was an internal failure when the Internet Security Manager attempted to map the resource URL to a security zone. In most default configurations, this method will return the zone `SECURITY_ZONE_INTERNET` for a valid URL unless the host has been explicitly added to the **Intranet Zone** or **Trusted Sites** list on the local system.

Security zones are used with Mark of the Web (MOTW) metadata which indicates if a file originated from an external source, such as the Internet or an untrusted network. When a file is downloaded using a web browser or other applications, it may add a `Zone.Identifier` alternate data stream to the file. This stream contains information such as `ZoneId=3`, which corresponds to the Internet Zone. Your application can use the **GetFileZoneInfo** method to retrieve this information, which allows you to determine whether a file was downloaded from an external source and what zone Windows has associated with it. This metadata affects how files are handled by Windows security components such as Microsoft Defender, SmartScreen, Office Protected View, and script execution policies in Windows Explorer. If the file has no metadata, it is generally treated as coming from the Local Machine Zone (zone 0). If your application sets or clears this metadata using the **SetFileZoneInfo** method, it can influence how other applications interpret the file's trust level.

It is important to note that this method is used to determine what security zone would be appropriate for a given resource, and it only performs syntactic checks and classification of the URL using the **IInternetSecurityManager** COM interface. The security zone this method returns is based on the current system and user policy, using zone assignments stored in the registry. Typically this information is managed using Windows settings or Internet options in the control panel. This method does not read the zone identifier metadata which has been assigned to a file.

For more detailed information about the zone identifiers, refer to the **SetDefaultZoneId** method.

Example

```
LPCTSTR pszDownloadUrl = _T("ftp://example.com/pub/download/file.txt");
DWORD dwZoneId = SECURITY_ZONE_NONE;

if (ftpClient.GetSecurityZoneId(pszDownloadUrl, &dwZoneId))
{
    // dwZoneId now contains SECURITY_ZONE_INTERNET
    _tprintf(_T("The security zone is: %u\n"), dwZoneId);
}
else
{
    DWORD dwError = ftpClient.GetLastError();
    // Handle the error
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultZoneId, GetFileZoneInfo, SetDefaultZoneId, SetFileZoneInfo, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetServerInformation Method

```
INT GetServerInformation(
   LPTSTR lpszSystemInfo,
   INT nMaxLength
);
```

The **GetServerInformation** method returns information about the server, typically including the operating system type, version and platform.

Parameters

lpszSystemInfo

The buffer that will contain the system information returned by the server.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the SYST command to the server. The first word will identify the type of operating system. The format for the remaining information depends on the server type. Typically it is a description of the operating system version and hardware platform.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetClientQuota, GetServerStatus, GetServerTimeZone, GetServerType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetServerStatus Method

```
INT GetServerStatus(
   LPTSTR lpszStatus,
   INT nMaxLength
);

INT GetServerStatus(
   CString& strStatus
);
```

The **GetServerStatus** method requests that the server return status information about itself.

Parameters

lpszStatus

The buffer that will contain the system status returned by the server. The alternate form of this method also accepts a **CString** object which will contain the system status string.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the STAT command to the server. The format for the information returned depends on the server type. Typically it is a description of the server platform, version, current user and file transfer options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetServerInformation, GetServerType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetServerTimeZone Method

```
INT GetServerTimeZone(
   LPLONG lpnTimeZone
);
```

The **GetServerTimeZone** method returns the timezone for the current server.

Parameters

lpnTimeZone

A pointer to a signed long integer which will contain the timezone offset in seconds.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This method sends the SITE ZONE command to the server to determine its timezone. The value returned is expressed as the number of seconds offset from Coordinated Universal Time (UTC). A positive value specifies a time west of UTC, while a negative value specifies a time east of UTC. For example, a value of 28800 would specify an offset of 8 hours west of UTC, which is the Pacific timezone.

The SIZE ZONE command is an extension that is not supported by all servers. If the server timezone cannot be determined, the method will fail and the value pointed to by the *lpnTimeZone* parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetClientQuota, GetServerInformation, GetServerStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetServerType Method

```
INT GetServerType();
```

The **GetServerType** method returns the type of server the client has connected to.

Parameters

None.

Return Value

If the method succeeds, the return value is a numeric value which indicates the server type. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the SYST command to the server to determine the server type. The following server types are recognized by the library:

Value	Description
FTP_SERVER_UNKNOWN	The server type could not be determined by issuing the SYST command. The server may not support the command, or the command may only be allowed when issued by an authenticated user.
FTP_SERVER_MSDOS	The server is running on an MS-DOS based operating system. The server expects file pathing and naming conventions according to the standard MS-DOS format and returns directory listings similar to the output of the DIR command.
FTP_SERVER_WINDOWS	The server is running on a Windows based operating system. The server expects file pathing and naming conventions according to the standard Windows long filename format, and returns directory listings similar to the output of the DIR command. Note that Windows servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a Windows platform.
FTP_SERVER_VMS	The server is running on a DEC VMS based operating system. The server expects file pathing and naming conventions specific to that operating system. Note that VMS servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a VMS platform.
FTP_SERVER_NETWARE	The server is running on a Netware based operating system. The server expects file pathing and naming conventions similar to the standard Windows long filename format, and returns directory listings that are similar to UNIX systems with the exception of the access and permissions flags for the file. Note that a Netware system may return listings in different formats based on the filesystem and site specific options specified.

FTP_SERVER_OTHER	The server type was not recognized. An attempt will be made to automatically determine the correct file pathing and naming conventions used by the server. To obtain a list of files on the server, it may be necessary to use the **SetDirectoryFormat** method to specify the directory listing format.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetClientQuota, GetServerInformation, GetServerStatus, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
FTP_STATUS_UNUSED	No connection has been established.
FTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
FTP_STATUS_CONNECT	The client is establishing a connection with the server.
FTP_STATUS_READ	The client is reading data from the server.
FTP_STATUS_WRITE	The client is writing data to the server.
FTP_STATUS_DISCONNECT	The client is disconnecting from the server.
FTP_STATUS_OPENFILE	The client is opening a data connection to the server.
FTP_STATUS_CLOSEFILE	The client is closing the data connection to the server.
FTP_STATUS_GETFILE	The client is downloading a file from the server.
FTP_STATUS_PUTFILE	The client is uploading a file to the server.
FTP_STATUS_FILELIST	The client is retrieving a file listing from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

IsBlocking, IsReadable, IsWritable, GetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetText Method

```
INT GetText(
   LPCTSTR lpszRemoteFile,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **GetText** method copies the contents of a text file on the server to the specified string buffer.

Parameters

lpszRemoteFile

A null-terminated string that specifies a text file on the server. The file pathing and naming conventions must be that of the host operating system.

lpszBuffer

A null-terminated string buffer which will contain the contents of the text file when the method returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetText** method is used to download a text file and store the contents in a string buffer. This method will always set the file type to FILE_TYPE_ASCII before downloading the file, and will restore the default file type before the method returns. Because binary files can include embedded null characters, this method should only be used with known text files.

This method has been included as a convenience for applications that need to retrieve relatively small text files and manipulate the contents as a string. If the Unicode version of this method is called, the contents of the text file is automatically converted to a Unicode string. If the size of the file is unknown or the text file is very large, it is recommended that you use the **GetData** or **GetFile** methods.

If you use the **GetFileSize** method to determine how large the string buffer should be prior to calling this method, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **GetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line characters.

FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = pClient->GetText(lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, EnableEvents, GetData, GetFile, GetTransferStatus, PutData, PutFile, RegisterEvent, SetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both commands issued on the command channel and file transfers using the data channel. The initial timeout value is set when establishing a connection and the default timeout period is 20 seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::GetTransferStatus Method

```
INT GetTransferStatus(
   LPFTPTRANSFERSTATUS lpStatus
);

INT GetTransferStatus(
   LPFTPTRANSFERSTATUSEX lpStatus
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an FTPTRANSFERSTATUS or FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this method will return the status of the last successful transfer made by the client.

The *dwBytesTotal* and *dwBytesCopied* members of the **FTPTRANSFERSTATUSEX** structure are declared as unsigned 64-bit integers rather than 32-bit integers. To obtain accurate file transfer information, this extended version of the structure should be used with files that are larger than 4GiB.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **FTPTRANSFERSTATUS** and **FTPTRANSFERSTATUSEX** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetFileStatus, GetStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Login Method

```
INT Login(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszAccount
);
```

The **Login** method authenticates the specified user in on the server. This method must be called after the connection has been established, and before attempting to transfer files or perform any other method on the server.

Parameters

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

lpszAccount

Points to a string that specifies the account name to be used to authenticate the current client session. This parameter may be NULL or an empty string if no account name is required for the specified user.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Some public FTP servers support anonymous logins, where a username and password are not required to access the server. In this case, both the *lpszUserName* and *lpszPassword* parameters can be NULL or specify empty strings. In most cases, access to the server using an anonymous login is restricted, with clients only having permission to download files. Servers may also restrict the maximum number of anonymous sessions that may be logged in at one time.

This method should only be used after calling the **Logout** method, enabling you to log in as another user during the same session. Not all servers will permit a client to change user credentials during the same session. In most cases, it is preferable to disconnect from the server and re-connect using the new credentials rather than using this method.

This method is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Logout, ProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Logout Method

```
INT Logout();
```

The **Logout** method logs out the user associated with the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Logout** method is used when the client wants to re-authenticate using a new username and password. Before any further action may be taken, other than disconnecting from the server, the **Login** method must be called to re-authenticate the client.

It is not necessary to call this method prior to disconnecting from the server because the user current user is automatically logged out when the **Disconnect** method is called.

This method is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::MountStructure Method

```
INT MountStructure(
   LPCTSTR lpszFileSystem
);
```

The **MountStructure** method mounts a different file system or other directory data structure on the server.

Parameters

lpszFileSystem

A null-terminated string which specifies the file system to mount on the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the SMNT command to the server, which may not be supported on some platforms. Use of this command typically requires that the user have administrator privileges on the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateDirectory, RemoveDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::NormalizePath Method

```
INT NormalizePath(
   LPCTSTR lpszFilePath,
   LPTSTR lpszNormalized,
   INT nMaxLength
);

INT NormalizePath(
   LPCTSTR lpszFilePath,
   CString& strNormalized
);
```

The **NormalizePath** method normalizes a file path using the current client connection.

Parameters

lpszFilePath

A null-terminated string which specifies a file path on the server. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized path when the method returns. The buffer must be large enough to contain the entire path including the terminating null character. If this parameter is NULL it will be ignored and the method will return the length of the normalized path without copying any characters. An alternate version of this method accepts a reference to a **CString** object which will contain the normalized path when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the method will return the length of the normalized path without copying any characters.

Return Value

This method returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized path but will not copy any characters. If the method fails, it will return FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will accept a relative or absolute file path on the server and return a normalized version of that path. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. For example, if the current working directory for the user is `/home/jsmith` and the file path provided is:

```
./text/..//data//./files/filename.txt
```

This method would return the following as the normalized path:

```
/home/jsmith/data/files/filename.txt
```

Because the path is relative, the current working directory for the client would be used to create an absolute path. The extraneous path delimiters are removed (the double slashes) and the reference to the parent directory is collapsed to produce a simple, uniform path. This can be useful both for display purposes and to normalize input provided by a user.

If a connection has not been established with a server, relative file paths will not be normalized as

absolute paths because there is no current working directory available. This allows the method to be used without an active connection, removing any extraneous path delimiters and collapsing the path without returning an absolute path unless one was provided.

If the path specified by the ***lpszFilePath*** parameter is relative to the current directory and there is no active connection, the normalized path returned by the method will always be relative. For example, if the path provided is:

```
../../../../../data/filename.txt
```

This method would return the following as the normalized path:

```
data/filename.txt
```

Because the original path was relative, the normalized path is also relative and will not begin with a leading path delimiter. It is intentionally designed this way to prevent the path from "escaping" to the root directory of a file system, which can present a security risk. If the original path was absolute, the normalized path would also be absolute.

This method will accept an FTP URL and return only the normalized path and file name potion of the URL. If you need to normalize a complete URL, use the **NormalizeUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFilePathUrl, NormalizeUrl, ValidateHostName ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::NormalizeUrl Method

```
INT NormalizeUrl(
   LPCTSTR lpszUrl,
   LPTSTR lpszNormalized,
   INT nMaxLength,
);

INT NormalizeUrl(
   LPCTSTR lpszUrl,
   CString& strNormalized
);
```

The **NormalizeUrl** method returns a normalized FTP URL.

Parameters

lpszUrl

A null-terminated string which specifies an FTP URL. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the method returns. The buffer must be large enough to contain the entire URL including the terminating null character. If this parameter is NULL it will be ignored and the method will return the length of the normalized URL without copying any characters. An alternate version of this method accepts a reference to a **CString** object which will contain the normalized URL when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the *lpszNormalized* parameter is ignored and the method will return the length of the normalized URL without copying any characters.

Return Value

This method returns the number of characters copied into the string buffer, not including the terminating null character. If the *lpszNormalized* parameter is NULL, it will return the length of the normalized URL but will not copy any characters. If the method fails, it will return FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will accept a standard FTP, FTPS or SFTP URL and return a normalized version of that URL. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. If non-standard port numbers are included in the URL, they will be preserved. For example, if the follwing URL is provided to the method:

```
ftps://ftp.example.com:21/file/..//data/filename.txt
```

It will return the following as the normalized URL:

```
ftpes://ftp.example.com/data/filename.txt
```

Because the original URL specified a secure connection on port 21, this is inferred to mean the connection will use explicit TLS and not an implicit connection. This is why the scheme is changed from `ftps://` to `ftpes://` and the port number is removed, since port 21 is the default port for an explicit TLS connection. In addition, the path is collapsed and the extraneous path delimiters are removed.

This method is primarily intended to normalize URLs which may be entered by a user or read from a configuration file. It is not necessary to call this method prior to calling **ConnectUrl** because it will always be normalized internally prior to the connection being established. If you wish to check a URL to ensure it is valid and the server name can be resolved to an IP address, use the **ValidateUrl** method.

If you need to normalize a remote file path, use the **NormalizePath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, GetFilePathUrl, NormalizePath, ValidateHostName ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::OpenDirectory Method

```
INT OpenDirectory(
   LPCTSTR lpszDirectory
);
```

The **OpenDirectory** method opens the specified directory on the server.

Parameters

lpszDirectory

Pointer to the name of the directory that will be opened. The format of the directory name must match the filename conventions used by the server. If a NULL pointer or an empty string is specified, then the current working directory is opened.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **OpenDirectory** method opens the specified directory on the server using the LIST command. The contents of the directory can be read using the **GetFirstFile** and **GetNextFile** methods. The directory listing is returned on the data channel in one of several different formats. The library can recognize listing formats generated by UNIX, VMS and Windows servers, as well as those of other servers which emulate one of those common formats. Once the complete directory listing has been read, the directory must be closed by calling the **CloseDirectory** method.

Because the directory listing is returned on the data channel, a file transfer cannot be performed while the directory is in the process of being read by the client. Applications which need to collect a list of files to download should first open the directory, read the contents and store the file names in an array. After the directory has been closed, the application can then start transferring the files to the local system.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **ChangeDirectory** method and then call this method again, passing NULL or an empty string as the *lpszDirectory* parameter.

To obtain a list of all files in a directory using a single function call, use the **EnumFiles** method. If the server lists files in a format that is not recognized by the library, the **GetFileList** method can be used to obtain an unparsed file listing from the server.

Example

```
if (pClient->OpenDirectory() != FTP_ERROR)
{
    FTPFILESTATUS ftpFile;
    BOOL bResult;

    bResult = pClient->GetFirstFile(&ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = pClient->GetNextFile(&ftpFile);
    }
```

```
    pClient->CloseDirectory();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, CloseDirectory, EnumFiles, GetDirectoryFormat, GetFileList, GetFileStatus, GetFirstFile, GetNextFile, SetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::OpenFile Method

```
INT OpenFile(
   LPCTSTR lpszFileName,
   DWORD dwOpenMode,
   DWORD dwOffset
);
```

The **OpenFile** method creates or opens the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to create or open. The file pathing and name conventions must be that of the server.

dwOpenMode

Specifies the type of access to the file. An application can open a file for reading, create a new file or append data to an existing file. This parameter should be one of the following values.

Value	Description
FTP_FILE_READ	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
FTP_FILE_WRITE	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
FTP_FILE_APPEND	The file is opened for writing on the server. All data will be appended to the end of the file.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error. Typically this indicates that the application failed to call the **CloseFile** method.

It is strongly recommended that most applications use the **GetFile** or **PutFile** methods to perform file transfers. These methods are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

When a file is created on the server, the file ownership and access rights are determined by the server. Some servers may provide a method to change these attributes through site-specific commands. Refer to the server's operating system documentation for more information about what commands may be available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFile, GetFile, GetFileSize, GetFileTime, PutFile, SetFileTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::ProxyConnect Method

```
HCLIENT ProxyConnect(
   UINT nProxyType,
   LPCTSTR lpszProxyHost,
   UINT nProxyPort,
   LPCTSTR lpszProxyUser,
   LPCTSTR lpszProxyPassword,
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **ProxyConnect** method establishes a connection through a proxy server.

Parameters

nProxyType

An identifier which specifies the type of proxy server that is being connected to. This value must be defined as one of the following values:

Value	Description
FTP_PROXY_NONE	This value specifies that no proxy server is being used. In this case, the **Connect** method is called directly, ignoring the proxy parameters.
FTP_PROXY_USER	This value specifies that the client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
FTP_PROXY_LOGIN	This value specifies that the client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
FTP_PROXY_OPEN	This value specifies that the client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
FTP_PROXY_SITE	This value specifies that the client is logged into the server. The SITE command is sent, specifying the host name, followed by the username and the password.
FTP_PROXY_OTHER	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the Command method to send specific command sequences.

lpszProxyHost

The name of the proxy server to connect through; this may be a fully-qualified domain name or an IP address.

nProxyPort

The port number the proxy server is listening on; a value of zero specifies that the default port number should be used.

lpszProxyUser

The user name used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a username is required.

lpszProxyPassword

The password used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a password is required.

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require

	authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when

	connecting to the server on ports other than 990.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when

	explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

Return Value

If the method succeeds, the return value is a handle to a client session. If the method fails, the return value is INVALID_CLIENT. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **ProxyConnect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The username and password that is used to authenticate the client with the proxy server are not the same as those used to login to the target server. Once a connection has been established with the proxy server, the client must call the **Login** method to actually login to the server and begin a file transfer.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will first send an AUTH TLS command to the server. If the server does not accept this command, it will then send an AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **SetChannelMode** method.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, ConnectUrl, CreateSecurityCredentials, Disconnect, GetSecurityInformation, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::PutData Method

```
INT PutData(
   LPCTSTR lpszRemoteFile,
   LPBYTE lpBuffer,
   DWORD dwLength
);

INT PutData(
   LPCTSTR lpszRemoteFile,
   LPCTSTR lpszBuffer
);
```

The **PutData** method transfers the contents of the specified buffer to a file on the server.

Parameters

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpBuffer

The data that will be copied to the server and stored in the specified file. An alternate version of the method uses a pointer to a string buffer where all of the bytes will be written to the server up to, but not including, the terminating null character.

dwLength

The number of bytes to copy from the buffer.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, EnableEvents, GetData, GetFile, GetTransferStatus, PutFile, RegisterEvent, SetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::PutFile Method

```
INT PutFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
   DWORD dwOffset
);
```

The **PutFile** method transfers the specified file on the local system to the server.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be uploaded. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be created, replaced or appended to on the local system. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, EnableEvents, GetData, GetFile, GetMultipleFiles, GetTransferStatus, PutData, PutMultipleFiles, RegisterEvent, SetBufferSize, VerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::PutMultipleFiles Method

```
INT PutMultipleFiles(
   LPCTSTR lpszLocalDirectory,
   LPCTSTR lpszRemoteDirectory,
   LPCTSTR lpszFileMask
);
```

The **PutMultipleFiles** method copies one or more files from the local host to the server, using the specified wildcard.

Parameters

lpszLocalDirectory

Pointer to a null-terminated string which specifies the local directory where the files will be uploaded from. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs. A NULL pointer or empty string specifies that files should be copied to the current working directory.

lpszRemoteDirectory

Pointer to a null-terminated string which specifies the remote directory where the files will be uploaded to. A NULL pointer or empty string specifies that the files should be copied to the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the local system to the server. The file mask should follow the Windows conventions used for wildcard file matches.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutMultipleFiles** method is used to transfer files from the local host to the server which match a specified wildcard file mask. All files will be transferred using the current file type as specified by the **SetFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, GetFile, GetMultipleFiles, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::PutText Method

```
INT PutText(
   LPCTSTR lpszRemoteFile,
   LPCTSTR lpszBuffer
);
```

The **PutText** method creates a text file on the server using the contents of a string buffer.

Parameters

lpszRemoteFile

A null-terminated string that specifies the text file on the server that will be created or overwritten. The file pathing and name conventions must be that of the server.

lpszBuffer

A null-terminated string that contains the text that will be stored in the file.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutText** method is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. This method will always set the file type to FILE_TYPE_ASCII before creating the file, and will restore the default file type before the method returns.

If the Unicode version of this method is called, the string will be converted to UTF-8 and then uploaded to the server. If you wish to store the contents of the string as UTF-16 Unicode on the server, you must set the current file type to FILE_TYPE_IMAGE and use the **PutData** method. This method should never be used to upload binary data.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, EnableEvents, GetText, GetTransferStatus, PutData, PutFile, RegisterEvent, SetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of the method accepts a **CString** object which will contain the data returned from the server.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetData, GetFile, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
FTP_EVENT_CONNECT	The control connection to the server has completed.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event occurs when a file download has completed. If multiple files

	are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event occurs when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.
FTP_EVENT_QUEUE	This event occurs during a queued file transfer. It only occurs when a file is being transferred using the **CFtpQueue** class and the internal state of the queue has changed. The handle passed to this event will be a handle to the queue, not a client connection to the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **FtpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a file is being uploaded or downloaded. The FTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one FTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The ***dwParam*** parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, GetTransferStatus, FtpEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::RegisterFileType Method

```
BOOL RegisterFileType(
   LPCTSTR lpszExtension,
   UINT nFileType
);
```

The **RegisterFileType** method associates a file name extension with a specific file type.

Parameters

lpszExtension

A null-terminated string which specifies the file name extension. If this parameter is NULL or points to an empty string, the default file type will be changed for the client session.

nFileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values.

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary image and no data conversion of any type is performed on the file. This is typically the default file type for data file transfers. If the type of file that is being transferred is unknown, this file type should always be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **RegisterFileType** method is used to associate specific file types with file name extensions. The library has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that list for the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAutoFileType, OpenFile, SetFileMode, SetFileStructure, SetFileType, SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::RemoveDirectory Method

```
INT RemoveDirectory(
   LPCTSTR lpszDirectory
);
```

The **RemoveDirectory** method removes the specified directory on the server.

Parameters

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the RMD command to create the directory. The user must have the appropriate permission to remove the specified directory. Most servers will not permit you to remove a directory if it contains one or more files.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeDirectory, CreateDirectory, DeleteFile, GetDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::RenameFile Method

```
INT RenameFile(
   LPCTSTR lpszOldFileName,
   LPCTSTR lpszNewFileName
);
```

The **RenameFile** method renames or moves the specified file on the server. The original file must exist and the current user must have the appropriate permissions to change the file name.

Parameters

lpszOldFileName

A null-terminated string which specifies the name of the remote file to rename or move. This parameter cannot be NULL or a zero-length string and the file must exist on the server. The file path and name must match the conventions used on the server.

lpszNewFileName

A null-terminated string which specifies the new name for the file. This parameter cannot be NULL or a zero-length string. The file path and name must match the conventions used on the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the *lpszNewFileName* parameter specifies a file which already exists on the server, the contents of that file will be deleted and replaced by the contents of the file specified by *lpszOldFileName*. If you want to prevent this from occurring, your application can check if a file with the new name exists by calling the **GetFileStatus** method. If successful, you can warn the user the rename operation would delete the contents of the new file as a consequence of renaming the original file.

If the connection was established using the standard FTP or FTPS protocols, this method causes two separate commands to be sent to the server, RNFR and RNTO. If either command fails, the method will fail and return an error code. If the connection was established using SFTP (SSH) the client will request the server rename the file atomically by specifying the internal SSH_FXF_RENAME_ATOMIC option. If the server does not support this option and *lpszNewFileName* already exists on the server, this method will attempt to delete the file and retry the rename operation.

This method can be used to move a file from one folder to another on the server, as long as the user has the appropriate permissions required to perform the operation. If the new file name includes a path, that path must already exist on the server or the method will fail.

If the old and new file names specify folders instead of regular files, the server may or may not allow the operation, depending on the access rights for those folders and the server's configuration. Some servers may allow a simple rename operation within the same parent folder, but may not permit you to move the folder to another location in the filesystem.

There is no guarantee the rename operation will be performed in an atomic fashion if multiple client sessions attempt to rename the same file at the same time. For example, it is possible that one client could attempt to rename a file while another client is already in the process of moving the file to a new folder. In this case, the server may respond with an error indicating the file cannot be found, an access denied error or a general failure error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteFile, GetFile, GetFileStatus, GetFileSize, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client cannot be reset while a file transfer is in progress or if the client is in a blocked state. To abort a file transfer, use the **Cancel** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetActivePorts Method

```
INT SetActivePorts(
   UINT nLowPort,
   UINT nHighPort
);
```

The **SetActivePorts** method changes the range of local port numbers used for active mode file transfers.

Parameters

hClient

Handle to the client session.

nLowPort

An unsigned integer that specifies the low port number.

lpnHighPort

An unsigned integer that specifies the high port number.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This method is used to modify the range of local port numbers used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **SetActivePorts** method can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

To determine the current range of active port numbers being used, call the **GetActivePorts** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetActivePorts, SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetBufferSize Method

```
INT SetBufferSize(
   INT nBufferSize
);
```

The **SetBufferSize** method sets the size in bytes of an internal buffer that will be used during data transfers.

Parameters

nBufferSize

The size of an internal buffer, in bytes. Any value greater than or equal to zero is acceptable. If ***nBufferSize*** is zero, then the default value of 4096 will be used. If ***nBufferSize*** is less than 256 bytes, the buffer size will be set to 256. The maximum value of ***nBufferSize*** is 1048576 (1Mb).

Return Value

If the method succeeds, the return value is the size of the internal buffer that will be used. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a multiple of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetBufferSize, GetData, GetFile, PutData, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetChannelMode Method

```
INT SetChannelMode(
    INT nMode
);

INT SetChannelMode(
   INT nChannel,
   INT nMode
);
```

The **SetChannelMode** method changes the security mode for the specified communication channel.

Parameters

nChannel

An integer value which specifies which channel to return information for. If the first version of this method is called, then the data channel mode is modified; otherwise, the channel may be explicitly specified as one of the following values:

Value	Description
FTP_CHANNEL_COMMAND	Change information for the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Change information for the data channel. This is the communication channel used to send or receive data during a file transfer.

nMode

An integer value which specifies the new mode for the specified channel. It may be one of the following values:

Value	Description
FTP_CHANNEL_CLEAR	Data sent and received on this channel should not be encrypted.
FTP_CHANNEL_SECURE	Data sent and received on this channel should be encrypted. Specifying this option requires that a secure connection has already been established with the server.

Return Value

If the method succeeds, the return value is the previous mode for the specified channel. If the method fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **SetChannelMode** method is used to change the default mode for the specified channel, and is typically used to control whether or not data is encrypted during a file transfer. If a standard, non-secure connection has been established with the server, an error will be returned if you specify the FTP_CHANNEL_SECURE mode for either channel.

If you have established a secure connection and then specify the FTP_CHANNEL_CLEAR mode for the command channel, the client will send the CCC command to the server to indicate that commands

should no longer be encrypted. If the server does not support this command, an error will be returned and the channel mode will remain unchanged. Once the command channel has been changed to clear mode, it cannot be changed back to secure mode. You must disconnect and re-connect to the server if you want to resume sending commands over an encrypted channel.

Changing the mode for the data channel requires that the server support the PROT command. If this command is not supported by the server, the method will fail and the channel mode will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetDefaultZoneId Method

```
BOOL SetDefaultZoneId(
   DWORD dwZoneId
);
```

The **SetDefaultZoneId** method sets the default security zone for downloaded files.

Parameters

dwZoneId

An unsigned integer which specifies the default security zone identifier. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it. This typically indicates that the file was created locally or has not been marked by Windows as originating from another zone. This is the default behavior if the application does not explicitly set a default security zone.
SECURITY_ZONE_LOCAL	The file is considered to be part of the **Local Machine** zone, which includes files stored on the user's local system or removable media. This zone has the least restrictive security settings, and files are typically allowed to run without prompting or limitation. ***This zone should not be assigned to downloaded files***, as doing so can bypass security mechanisms designed to protect against potentially harmful content. Only use this setting for files that are created locally by the application itself and are known to be safe. Misuse of this setting can result in a significant security risk, especially if the file contains scripts, executables, or other active content.
SECURITY_ZONE_INTRANET	The file originates from the **Local Intranet** zone, which includes content served from internal network resources, such as shared drives, file servers, and internal web applications. This zone typically has fewer security restrictions than the Internet zone but more than the Local Machine zone. It is commonly used in enterprise environments where network resources are managed and considered relatively safe. ***This zone should not be used for files downloaded from the public Internet***, as doing so may incorrectly signal that the file is trusted within the organization. Assign this zone only to files that are retrieved from authenticated internal sources or known secure locations on the local network.
SECURITY_ZONE_TRUSTED	The file is from a site or source explicitly added to the **Trusted Sites** zone in Internet Explorer or Windows security settings. This zone is intended for domains or locations that the user or system administrator considers

	to be safe, such as internal applications hosted on the public Internet or third-party services with a strong reputation for safety. Files marked with this zone typically bypass certain security checks, and applications may allow active content (such as scripts or macros) to run without prompting. Use caution when assigning this zone to downloaded files, as it may override protections that would otherwise apply to untrusted content.
SECURITY_ZONE_INTERNET	The file was downloaded from the **Internet** zone, which represents untrusted or unknown sources outside the local network. This is the standard zone identifier automatically applied by web browsers, email clients, and many applications when a file is downloaded from a public website or external server. Windows uses this zone to enforce security restrictions, such as prompting the user before executing scripts, blocking active content, or displaying a warning when the file is opened. This is typically the appropriate zone to assign for any file obtained over FTP from an external site. Assigning this zone helps ensure that users are notified before opening potentially unsafe content, in accordance with Windows Attachment Manager and Group Policy settings.
SECURITY_ZONE_RESTRICTED	The file is associated with the **Restricted Sites** zone, which enforces the strictest security policies. This zone is used for content that is explicitly marked as untrusted or potentially harmful. Files tagged with this zone may be blocked from execution, prevent active content from running, or trigger warnings when accessed. Assigning this zone can be useful for sandboxing unverified downloads or limiting the risk from files obtained from questionable sources. However, it may also cause legitimate files to fail to open or operate with reduced functionality depending on system policy and application behavior.

Return Value

If the method succeeds, the return value is non-zero. If the method fails it will return zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SetDefaultZoneId** method sets the default security zone identifier that will be applied to all files downloaded during the client session. This setting controls how Windows treats downloaded files with respect to security prompts and content execution restrictions. Once the default zone ID is set, it will be automatically applied to all subsequent file downloads performed by the client session. This eliminates the need to manually set the security zone for each individual file, ensuring consistent zone tagging across the session.

When `SECURITY_ZONE_NONE` is specified, no `Zone.Identifier` alternate data stream is created. This causes the file to be treated as if it originated from the local system, and Windows will not apply zone-based security restrictions. This is the default behavior which maintains backwards compatibility with previous versions of the library that did not create the security zone metadata for files.

The zone ID metadata is interpreted by Windows components such as **Attachment Execution Services** and **SmartScreen** to determine whether the user should be warned when opening a file. In enterprise environments, Group Policy settings may alter how different zone IDs are handled, such as blocking scripts, macros, or unsigned executables. Developers should test zone assignment behavior in target environments when distributing files to end users.

Note that this method does not retroactively modify existing files. The default zone identifier is applied only at the time a file is downloaded using methods such as **DownloadFile**.

If your application is distributed in a controlled environment, such as a corporate network with endpoint protection or user access controls, consult your organization's security policy before modifying default zone behavior. Improper zone tagging can impact file access, generate security prompts, or interfere with digital signature enforcement.

Be aware that some file operations (such as extracting from a ZIP archive, or copying files to non-NTFS volumes or across certain network shares) may strip the `Zone.Identifier` metadata. Developers should not rely solely on zone tagging as a persistent trust mechanism outside of the initial download context.

To get the security zone identifier for a local file, use the **GetFileZoneInfo** method. To change or remove the security zone information for a file, use the **SetFileZoneInfo** method.

Assigning `SECURITY_ZONE_LOCAL` as the default security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetDefaultZoneId, GetFile, GetFileZoneInfo, SetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetDirectoryFormat Method

```
INT SetDirectoryFormat(
   INT nFormatId
);
```

The **SetDirectoryFormat** method is used to specify the format used by the server when returning a list of files. The format type is used internally by the library when parsing the file list returned by the server.

Parameters

nFormatId

An identifier used to specify the format of the file list returned by the server. The following values are recognized:

Value	Description
FTP_DIRECTORY_AUTO	This value specifies that the library should automatically determine the format of the file lists returned by the server. It is recommended that most applications use this value and allow the library to automatically determine the appropriate file listing format used by the server.
FTP_DIRECTORY_UNIX	This value specifies that the server returns file lists in the format commonly used by UNIX servers. Note that many servers can be configured to return file listings in this format, even if they are not actually a UNIX based platform. Consult the technical reference documentation for your server for more information.
FTP_DIRECTORY_MSDOS	This value specifies that the server returns file lists in the format commonly used by MS-DOS based systems. This includes Windows IIS servers. Long file names will be returned if supported by the underlying filesystem, such as NTFS or FAT32.
FTP_DIRECTORY_VMS	This value specifies that the server returns file lists in the format commonly used by VMS servers. Note that VMS servers can be configured to return a standard UNIX style listing in additional to the default VMS format.
FTP_DIRECTORY_STERLING_1	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 13 byte status code.
FTP_DIRECTORY_STERLING_2	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 10 byte status code.
FTP_DIRECTORY_NETWARE	This value specifies that the server returns file listings in a proprietary format used by Netware servers. The format is similar to UNIX style listings except that file access and

	permissions are indicated by letter codes enclosed in brackets. This is the default format selected if the server identifies itself as a Netware system.
FTP_DIRECTORY_MLSD	This value specifies that the server should return file listings in a machine-independent format as defined by RFC 3659. This format specifies file information as a sequence of name/value pairs, with the same format being used regardless of the operating system that the server is hosted on. Note that not all servers support this format, and some proxy servers may reject the command even if the remote server supports its use.

Return Value

If the method succeeds, the return value identifies the file list format used by the server. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should only be used when the library cannot automatically determine the directory format returned by the server. To determine the format used by a server after a file list has been retrieved, use the **GetDirectoryFormat** method.

The default directory format is determined both by the server's operating system and by analyzing the format of the data returned by the server. If the library is unable to automatically determine the format, it will attempt to parse the list of files as though it is a UNIX style listing.

If the FTP_DIRECTORY_MLSD format is specified, the file information returned by the server may differ from the default output of the LIST command. For example, on a UNIX based FTP server, the output of the LIST command is typically the same format that is used by the /bin/ls command, where file names are sorted and hidden files are not listed. However, the MLSD command may return an unsorted list of files that includes hidden files and directories.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseDirectory, GetDirectoryFormat, GetFileStatus, GetFirstFile, GetNextFile, OpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFeatures Method

```
DWORD SetFeatures(
   DWORD dwFeatures
);
```

The **SetFeatures** method specifies the server features available to the client.

Parameters

dwFeatures

An unsigned integer that specifies one or more features. Refer to the documentation for the **GetFeatures** method for a list of available features.

Return Value

If the method succeeds, the return value specifies the features that were previously enabled. If the method fails, it will return zero. Because it is possible that no features were enabled, a return value of zero does not always indicate an error. An application should call **GetLastError** to determine if an error code has been set.

Remarks

The **SetFeatures** method is used to enable a specific set of features for the current session. When a client connection is first established, all features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. To enable or disable a specific feature, an application can use the **EnableFeature** method.

For example, the first time an application calls the **GetFileSize** method to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

EnableFeature, GetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFileMode Method

```
INT SetFileMode(
   UINT nMode
);
```

The **SetFileMode** method sets the default file transfer mode for the current client session.

Parameters

nMode

Specifies the default type of data transfer mode for files being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_MODE_STREAM	The data is transmitted as a stream of bytes. This is the default client transfer mode.
FILE_MODE_BLOCK	The data is transmitted as a series of data blocks preceded by one or more header bytes. This transfer mode is currently not supported.
FILE_MODE_COMPRESSED	The data is transmitted in compressed form. This transfer mode is currently not supported.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The file transfer mode should be set before a file is opened or created on the server. Once the transfer mode is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OpenFile, SetFileStructure, SetFileType, SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFileNameEncoding Method

```
INT SetFileNameEncoding(
   INT nEncoding
);
```

The **SetFileNameEncoding** method specifies what type of encoding will be used when file names are sent to the server.

Parameters

nEncoding

An integer value which specifies the encoding type. It may be one of the following values:

Value	Description
FTP_ENCODING_ANSI	File names are sent as 8-bit characters using the default character encoding for the current codepage. If the Unicode version of the functions are used, file names are converted from Unicode to ANSI using the current codepage before being sent to the server. This is the default encoding type.
FTP_ENCODING_UTF8	File names that contain non-ASCII characters are sent using UTF-8 encoding. This encoding type is only available on servers that advertise support for UTF-8 encoding and permit that encoding type to be enabled by the client.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **SetFileNameEncoding** method can be used to enable UTF-8 encoding of file names, which provides improved support for the use of international character sets. However, the server must provide support for UTF-8 encoding by advertising it in response to the FEAT command and it must support the OPTS command which is used to enable UTF-8 encoding. If the server does not advertise support for UTF-8, or the OPTS command fails with an error, then this method will fail with an error and the encoding type will not change.

Although it is possible to use the **EnableFeature** method to explicitly enable the FTP_FEATURE_UTF8 feature, this is not recommended. If the server has not advertised support for UTF-8 encoding in response to the FEAT command, that typically indicates that UTF-8 encoding is not supported. Attempting to force UTF-8 encoding can result in unpredictable behavior when file names contain non-ASCII characters.

It is important to note that not all FTP servers support UTF-8 encoding, and in some cases servers which advertise support for UTF-8 encoding do not implement the feature correctly. For example, a server may allow a client to enable UTF-8 encoding, but once enabled will not permit the client to disable it. Some servers may advertise support for UTF-8 encoding, however if the underlying file system does not support UTF-8 encoded file names, any attempt to upload or download a file may fail with an error indicating that the file cannot be found or created.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Command, EnableFeature GetFeatures, GetFileNameEncoding, SetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFilePermissions Method

```
INT SetFilePermissions(
   LPCTSTR lpszFileName,
   DWORD dwPermissions
);
```

The **SetFilePermissions** method returns information about the access permissions for a specific file on the server.

Parameters

lpszFileName

A null-terminated string which contains the name of the file to be updated. The filename cannot contain any wildcard characters.

dwPermissions

An unsigned integer which will specify the new access permissions for the file. The file permissions are represented as bit flags, and may be one or more of the following values combined with a bitwise Or operator:

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.

FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the method succeeds, the return value is a result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the SITE CHMOD command to set the permissions for the file. This command is typically only supported on servers that are hosted on UNIX based systems. If the command is not supported, an error will be returned.

Users who are familiar with the UNIX operating system will recognize the **chmod** command used to change the file permissions. However, it should be noted that the numeric value used as an argument to the command is in octal, not decimal. For example, issuing the command **chmod 644 filename.txt** on a UNIX based system will make the file readable and writable by the owner, and readable by other users in the owner's group as well as all other users. The value 644 is an octal value, which is equivalent to the decimal value 420. If you were to mistakenly specify 644 as the value for the ***dwPermissions*** parameter, rather than the decimal value of 420, the permissions on the file would be incorrect. It is strongly recommended that you use the pre-defined constants to prevent this sort of error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFilePermissions, GetFileStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFileStructure Method

```
INT SetFileStructure(
    UINT nType
);
```

The **SetFileStructure** method sets the default file structure for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

nType

Specifies the default type of file structure being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_STRUCT_NONE	The file has no inherent structure and is considered to be a stream of bytes. This is the default structure for file transfers.
FILE_STRUCT_RECORD	The file uses a record structure. This file structure is currently not supported.
FILE_STRUCT_PAGE	The file uses a page structure. This file structure is currently not supported.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The file structure should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OpenFile, SetFileMode, SetFileType, SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFileTime Method

```
INT SetFileTime(
   LPCTSTR lpszFileName,
   LPSYSTEMTIME lpFileTime
);
```

The **SetFileTime** method sets the modification time for the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a SYSTEMTIME structure that specifies the new modification time for the remote file.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetFileTime** method will change the modification time of a file on the server. The values specified in the SYSTEMTIME structure are expected to represent UTC time, not time adjusted for the local system's timezone. If the values do represent the local time, it must be converted to UTC time prior to calling this method. To populate the SYSTEMTIME structure with the current time, use the **GetSystemTime** method.

When connected to an FTP server, this method uses the MDTM command to set the modification time for the specified file. Not all servers implement this command, in which case the method call will fail. Note that some servers only support the MDTM command to return, but not change, the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileStatus, GetFileTime, OpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFileType Method

```
INT SetFileType(
   UINT nFileType
);
```

The **SetFileType** method sets the default file type for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

hClient

Handle to the client session.

nFileType

Specifies the default type of file being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **FtpRegisterFileType** function.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The file type should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created. Some methods, such as **OpenDirectory** and **GetText**, will temporarily change the default file type to FILE_TYPE_ASCII and then restore the current file type when they return.

Calling this method has no practical effect when connected to an SFTP (SSH) server. They do not differentiate between text and binary files and the default file type will always be FILE_TYPE_IMAGE. If your application is uploading or downloading a text file, this difference between FTP and SFTP is important because the operating system that hosts the server may have different end-of-line character conventions than the client system. For example, if you download a text file from a UNIX system using SFTP, the end-of-line is indicated by a single linefeed (LF) character. However, on the Windows platform, the end-of-line is indicated by a carriage-return and linefeed sequence (CRLF).

The **GetFileType** method can be used to determine the current file type. To determine the transfer file type for a specific file, use the **GetAutoFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetAutoFileType, GetFileType, OpenFile, RegisterFileType, SetFileMode, SetFileStructure, SetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetFileZoneInfo Method

```
BOOL SetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   DWORD dwZoneId,
   LPCTSTR lpszHostUrl
);
```

The **SetFileZoneInfo** method sets or clears the zone identifier information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and write to the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

dwZoneId

An unsigned integer which specifies the zone ID which should be assigned to the file. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string which specifies the URL used to originally download the file. If this information is not required the parameter may be a NULL pointer or an empty string. If an originating URL is provided, it must be a valid FTP URL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails it will return zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SetFileZoneInfo** method enables applications to set or remove the security zone identifier metadata associated with a local file. This information is stored in an alternate data stream named `Zone.Identifier` and is used by Windows to determine the origin and trust level of a file, potentially prompting the user with a warning before the file is opened.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This

includes UNC paths (such as **\\server\share\file.txt**) and drive letters that are mapped to network shares. Security zone metadata cannot be applied to files on network volumes because the SMB protocol does not support alternate data streams. If a file with zone metadata is copied from a local NTFS drive to a network share, any associated streams like **Zone.Identifier** will be lost.

Specifying **SECURITY_ZONE_NONE** will remove any existing zone identifier stream from the file. This has the same effect as selecting the Unblock option in the file's properties dialog in Windows Explorer. Once the stream is removed, the file will be treated as though it originated from the local computer. Note that when the **Zone.Identifier** stream is deleted from a file to remove its security zone information, Windows Explorer may not immediately reflect this change. This is because Explorer caches certain file metadata, including zone information used to display the "This file came from another computer" warning in the file's Properties dialog.

When assigning a zone ID, it is important to ensure that the value accurately reflects the source of the file. For files that have been downloaded from the Internet, it is strongly recommended to use **SECURITY_ZONE_INTERNET**. This allows Windows and other software to enforce appropriate security measures, such as requiring user confirmation before executing or opening the file.

In most scenarios, zone identifiers should only be set for files that were downloaded from untrusted or external sources. Setting an overly permissive zone (such as **SECURITY_ZONE_LOCAL**) for files obtained from the Internet or other potentially unsafe locations can expose users to security risks and circumvent built-in protections.

If a URL is provided via the *lpszHostUrl* parameter, it will be included in the zone metadata and can be retrieved later using the **GetFileZoneInfo** method. This provides additional context about the origin of the file and may aid in auditing or diagnostic tasks. The URL must be a syntactically valid FTP address and should reflect the actual location from which the file was downloaded.

Not all file systems support alternate data streams. The method will fail if used on a volume (such as FAT32) that does not support this feature. Additionally, modifying zone information may fail if the file is in use, read-only, or if the current process does not have the required permissions. It is possible to disable the creation of security zone metadata through local policy settings. In this case, the method will fail and the last error code will be set to **ST_ERROR_POLICY_NO_ZONEINFO**.

Assigning **SECURITY_ZONE_LOCAL** as the security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultZoneId, GetFile, GetFileZoneInfo, SetDefaultZoneId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetIdleTimeout Method

```
INT SetIdleTimeout(
   INT nIdleTime
);
```

The **SetIdleTimeout** method sets the server-side idle timeout period for the control connection.

Parameters

nIdleTime

The number of seconds until the control session is considered inactive. A value of zero specifies the default idle timeout period of 300 seconds (5 minutes) will be used. If the value is less than 30 seconds or greater than 7200 seconds, it will be automatically adjusted to fit within the valid range.

Return Value

If the method succeeds, the return value is the previous idle timeout period. If the method fails, the return value is FTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **SetIdleTimeout** method changes the idle timeout period for the control connection, which can be useful in scenarios where a file transfer session is expected to remain idle for extended periods between commands. For example, when transferring large batches of files or performing periodic synchronization, the server may automatically close the connection if no activity is detected within its default timeout period. Increasing the idle timeout helps prevent unexpected disconnects during these idle periods, reducing the need to repeatedly reconnect and re-authenticate. Conversely, reducing the timeout can be used to enforce shorter sessions in security-sensitive environments.

The `SITE IDLE` command is a non-standard extension to the FTP protocol that allows a client to query or set the server's idle timeout period for the current session. When supported, it enables the client to request how long the server should wait during periods of inactivity before automatically closing the connection. This setting does not affect the duration of data transfers, only how long the control session is kept open when idle.

Because `SITE IDLE` is not part of the official FTP specification, support for it varies between servers. If the command is not supported, the method will fail but the session will remain connected using the server's default timeout settings. The method will also return an error if a negative timeout value is specified or if the session is using SFTP, which does not support this operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Command, GetIdleTimeout, GetTimeout, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetPassiveMode Method

```
INT SetPassiveMode(
   BOOL bPassiveMode
);
```

The **SetPassiveMode** method enables or disables passive mode file transfers for the specified client session.

Parameters

bPassiveMode

A boolean flag which specifies that the client should enter passive mode and establish all connections with the server to transfer data.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

By default, the File Transfer Protocol uses active mode transfers, whereby the data connection is established from the server back to the local client. However, this can introduce problems for a client application that is behind a proxy server, firewall or a router which uses Network Address Translation (NAT). Enabling passive mode transfers instructs the client to create an outbound connection from the local system to the server for the data connection, similarly to how the control connection is established.

Not all servers may support passive mode, in which case an error will be returned to the client when this method is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetData, GetFile, ProxyConnect, PutData, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetPriority Method

```
INT SetPriority(
   INT nPriority
);
```

The **SetPriority** method specifies the priority for file transfers.

Parameters

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Value	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the method succeeds, the return value is the previous file transfer priority. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetPriority** method can be used to control the processor usage, memory and network bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the

process, the FTP_PRIORITY_BACKGROUND value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The FTP_PRIORITY_CRITICAL priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to FTP_PRIORITY_CRITICAL, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete. A value of zero specifies the default timeout period of 20 seconds should be used.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both commands issued on the command channel and file transfers using the data channel.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** function in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::UploadFile Method

```
BOOL UploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPFTPTRANSFERSTATUS lpStatus
   FTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **UploadFile** method uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be uploaded. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be

	necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **UploadFile** method provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then upload the file to the server. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, then function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be uploaded as a text or binary file. A value of "a" specifies that the file should be uploaded as a text file. A value of "i" specifies that the file should be uploaded as a binary file. If the type is not explicitly specified, the file will be uploaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **UploadFile** method is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **PutFile** to upload the file.

Example

```
CFtpClient ftpClient;
CString strLocalFile = _T("c:\\temp\\database.mdb");
CString strFileURL =
_T("ftp://update:secret@ftp.example.com/updates/database.mdb");

if (!ftpClient.UploadFile(strLocalFile, strFileURL))
{
    ftpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, DownloadFile, GetFileZoneInfo, GetTransferStatus, PutFile, SetFileZoneInfo, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ValidateHostName Method

```
BOOL ValidateHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszAddress,
   INT nMaxLength
);

BOOL ValidateHostName(
   LPCTSTR lpszHostName,
   CString& strAddress
);
```

The **ValidateHostName** method determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The method will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the ***nMaxLength*** parameter is zero, it will be ignored and the IP address will not be returned. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the ***lpszAddress*** string buffer. The buffer must be large enough to store the complete address. Because this method can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the ***lpszAddress*** parameter will be ignored.

Return Value

If the method succeeds, the host name is valid and the return value will be non-zero. If the method fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ValidateHostName** method provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. If the Unicode version of this method is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

If the ***lpszHostName*** parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address.

If you wish to validate a complete FTP URL instead of a host name, use the **ValidateUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::ValidateUrl Method

```
BOOL ValidateUrl(
   LPCTSTR lpszUrl
);
```

The **ValidateUrl** method determines if a string represents a valid FTP URL.

Parameters

lpszUrl

A null-terminated string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **ValidateUrl** method will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure FTP scheme. This method will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this method will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **ConnectUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, DownloadFile, UploadFile, ValidateHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::VerifyFile Method

```
BOOL VerifyFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
);
```

The **VerifyFile** method attempts to verify that the contents of a file on the local system are the same as the specified file on the server.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system the remote file will be validated against. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be validated. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

Specifies the options that may be used when comparing the files. This parameter may be one or more of the following values:

Value	Description
FTP_VERIFY_DEFAULT	File verification should use the best option available based on the available server features. If the server supports the XSHA1 or XMD5 command, the method will calculate an SHA-1 or MD5 hash of the local file contents and compare the value with the file on the server. If the server does not support either hash algorithm, but it does support the XCRC command, the method will calculate a CRC32 checksum of the local file contents and compare the value with the file on the server. If the server does not support any hash or checksum commands, the method will compare the size of the local and remote files.
FTP_VERIFY_SIZE	Files are verified by comparing the number of bytes of data in the local and remote files. This is the least reliable option, and should only be used if the server does not support either the XSHA1, XMD5 or XCRC commands.
FTP_VERIFY_CRC32	Files are verified by calculating a CRC32 checksum of the local file contents and comparing it with the value returned by the server in response to the XCRC command. This option should only be used if the server does not support the XSHA1 or XMD5 commands.
FTP_VERIFY_MD5	Files are verified by calculating an MD5 hash of the local file contents and comparing it with the value returned by the server in response to the XMD5 command. This option should only be used

	if the server does not support the XSHA1 command.
FTP_VERIFY_SHA1	Files are verified by calculating an SHA-1 hash of the local file contents and comparing it with the value returned by the server in response to the XSHA1 command. This is the preferred option for performing file verification.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **VerifyFile** method will attempt to verify that the contents of the local and remote files are identical using one of several methods, based on the features that the server supports. Preference will be given to the most reliable method available, using either an MD5 hash, a CRC32 checksum or comparing the size of the file, in that order.

The method used to verify a file's contents determines how confidently it can be said that the local and remote files are identical. This method supports several algorithms, listed below in order of preference based on reliability and availability:

- SHA-1 is a cryptographic hash function that produces a 160-bit (20-byte) hash value. It is the most reliable option supported by this function and is preferred for verifying file contents. Although no longer recommended for cryptographic security due to potential collision attacks, it remains well-suited for file integrity checks where the threat of intentional tampering is low.
- MD5 generates a 128-bit (16-byte) hash and is faster than SHA-1. It offers a reasonable balance between performance and reliability for non-security-critical applications such as file verification. However, it is less resistant to collisions and should be used only if SHA-1 is not supported by the server.
- CRC32 is a non-cryptographic checksum algorithm that produces a 32-bit value. It is designed for detecting accidental data corruption rather than verifying content integrity. CRC32 is significantly faster but also far more prone to collisions, and should only be used when stronger methods like SHA-1 or MD5 are unavailable.
- If none of the supported hash or checksum commands are available, the function falls back to comparing the local and remote file sizes. This is the least reliable method and can only detect differences in file length, not in actual content.

It is not recommended that you use this method with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **VerifyFile** method to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetFileSize, GetFileStatus, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer,
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string is passed as the buffer argument. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Data Structures

- FTPCLIENTQUOTA
- FTPFILESTATUS
- FTPFILESTATUSEX
- FTPTRANSFERSTATUS
- FTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPCLIENTQUOTA

The **FTPCLIENTQUOTA** structure provides information about the file quota for the current client session.

```
typedef struct _FTPCLIENTQUOTA
{
    DWORD   dwFileCount;
    DWORD   dwFileLimit;
    DWORD   dwDiskUsage;
    DWORD   dwDiskLimit;
} FTPCLIENTQUOTA, *LPFTPCLIENTQUOTA;
```

Members

dwFileCount

An unsigned integer value which specifies the number of files that the user has created. If file quotas have not been enabled for the current user, this value will be zero.

dwFileLimit

An unsigned integer value which specifies the maximum number of files that may be created by the user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskUsage

An unsigned integer value which specifies the number of bytes of disk storage that has been allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskLimit

An unsigned integer value which specifies the maximum number of bytes of disk storage that may be allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPFILESTATUS

This structure is used by the **EnumFiles**, **GetFirstFile** and **GetNextFile** methods to return information about files on the server.

```
typedef struct _FTPFILESTATUS
{
    TCHAR        szFileName[FTP_MAXFILENAMELEN];
    TCHAR        szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR        szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL         bIsDirectory;
    DWORD        dwFileSize;
    DWORD        dwFileLinks;
    DWORD        dwFileVersion;
    DWORD        dwFilePerms;
    SYSTEMTIME   stFileDate;
} FTPFILESTATUS, *LPFTPFILESTATUS;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

dwFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify the

individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the chmod command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

stFileDate

A SYSTEMTIME structure which specifies the date that the file was created or last modified.

File Permissions

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value

A	0	1h
B	1	2h
C	2	4h
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPFILESTATUSEX

This structure is used by the **EnumFiles**, **GetFirstFile** and **GetNextFile** methods to return information about files on the server. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPFILESTATUSEX
{
    TCHAR          szFileName[FTP_MAXFILENAMELEN];
    TCHAR          szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR          szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL           bIsDirectory;
    ULARGE_INTEGER uiFileSize;
    DWORD          dwFileLinks;
    DWORD          dwFileVersion;
    DWORD          dwFilePerms;
    DWORD          dwFileFlags;
    SYSTEMTIME     stFileDate;
} FTPFILESTATUSEX, *LPFTPFILESTATUSEX;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

uiFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the chmod command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

dwFileFlags

This structure member is reserved for future use.

stFileDate

A SYSTEMTIME structure which specifies the date that the file was created or last modified.

File Permissions

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	0	1h
B	1	2h
C	2	4h
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUS

This structure is used by the **GetTransferStatus** method to return information about a file transfer in progress.

```
typedef struct _FTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
   TCHAR   szLocalFile[FTP_MAXFILENAMELEN];
   TCHAR   szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUS, *LPFTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

szLocalFile

A null-terminated string which specifies the local file that is being copied to or from the server.

szRemoteFile

A null-terminated string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FTPTRANSFERSTATUSEX** structure which uses 64-bit integers for the file size.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

GetTransferStatus, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUSEX

This structure is used by the **GetTransferStatus** method to return information about a file transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPTRANSFERSTATUSEX
{
   ULARGE_INTEGER dwBytesTotal;
   ULARGE_INTEGER dwBytesCopied;
   DWORD          dwBytesPerSecond;
   DWORD          dwTimeElapsed;
   DWORD          dwTimeEstimated;
   DWORD          dwReserved;
   TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
   TCHAR          szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUSEX, *LPFTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

dwReserved

This structure member is reserved for future use.

szLocalFile

A null-terminated string which specifies the local file that is being copied to or from the server.

szRemoteFile

A null-terminated string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h

See Also

GetTransferStatus, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname

used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A null-terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SFTP (SSH) connections and should always be NULL when establishing a secure connection using FTPS (FTP+TLS). If the member is NULL, then no private key is used. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that

it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The ***lpszKeyFile*** path can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results with multithreaded applications because the current working directory for a process is a global value and it can be changed by any thread at any time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol was selected. This protocol will only be used when connecting to an SSH server and is not supported with any other application protocol. Version 1.0 of the protocol is no longer considered secure and is not supported.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description

SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the TLS

protocol.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the TLS protocol.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the TLS protocol.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Queue Class Library

Manage a queue which will perform file transfers in the background.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CFtpQueue
File Name	CSFTPV12.DLL
Version	12.0.3024.1286
LibID	A86C8154-EC75-4C39-BB05-B698A87E98F1
Import Library	CSFTPV12.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

The File Transfer Queue class library provides a high-level interface to uploading and downloading a collection of files. Queued file transfers are performed in a background worker thread, with optional event notifications to inform the application about the current state of the queue and the files being processed.

A file may be queued for upload or download, and both uploads and downloads may be included in the same queue. Multiple instances of this class may be created to transfer multiple files at the same time. This class supports the same options available to individual file transfers using the CFtpClient class.

This class supports active and passive mode file transfers, firewall compatibility options, proxy servers and secure file transfers using TLS 1.3 and SFTP (SSH 2.0). Secure file transfers support implicit and explicit TLS sessions, client certificates and 256-bit AES encryption. SFTP transfers support public/private key authentication using either RSA or OpenSSH formats.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum

required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Queue Class Methods

Class	Description
CFtpQueue	Constructor which initializes the current instance of the class
~CFtpQueue	Destructor which releases resources allocated by the class
Method	Description
AddFile	Add a file to the transfer queue
AttachHandle	Attach the specified client handle to this instance of the class
Cancel	Cancel the specified queued file transfer
Clear	Remove all files from the transfer queue
DetachHandle	Detach the handle for the current instance of this class
DisableTrace	Disable logging of network function calls
EnableTrace	Enable logging of network function calls to a file
EnumFiles	Enumerate queued files based on the specified criteria
FindFile	Find a specific file in the queue
FtpEventProc	Callback function that processes events generated during queued transfers
GetClient	Return a handle to the active client session during a queued file transfer
GetCount	Return the number of files queued for transfer
GetErrorString	Return a description for the specified error code
GetFile	Return information about the specified file
GetFirstFile	Return information about the first file in the transfer queue
GetHandle	Return a handle for the queue
GetLastError	Return the last error code
GetNextFile	Return information about the next file in the transfer queue
GetStatus	Return the current status of the file transfer queue
GetTransferStatus	Return information about the file being currently transferred
IsEmpty	Determine if the current file transfer queue is empty
IsIdle	Determine if the queue manager is currently idle
IsInitialized	Determine if the class has been successfully initialized
RegisterFileType	Associate a file name extension with a specific file type
RemoveFile	Remove a file from the transfer queue
Reset	Reset the internal state of file transfers in the queue
Resume	Resume the transfer of files after queue processing has been paused
SetLastError	Set the last error code
ShowError	Display a message box with a description of the specified error

Start	Begin transferring files in the queue
Stop	Stop transferring files in the queue
Suspend	Pause the transfer of files in the queue
Wait	Wait for the transfer of all queued files to complete

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::CFtpQueue Method

```
CFtpQueue();

CFtpQueue(
   UINT nMaxFiles,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **CFtpQueue** constructor initializes the class library and validates the license key at runtime.

Parameters

nMaxFiles

An unsigned integer which specifies the maximum number of files which can be added to the queue. The constant INFINITE can be used to specify that there is no fixed limit to the number of files in the queue.

nTimeout

An unsigned integer which specifies the default timeout for all queued file transfers. If this value is zero, a reasonable default timeout period will be used. This timeout period is used when a file is added to the queue without providing a timeout period for that specific file transfer. If a timeout period is specified for a particular file, it will override this value.

dwOptions

An unsigned integer value which specifies one or more default options for all queued files. This parameter is constructed by using a bitwise operator and can be any of the options used with the **CFtpClient::Connect** method. If transfer options are specified for a particular file, it will override this value. The most common options are:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for queued file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies queued transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies queued transfers should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs

	Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies queued file transfers should attempt to establish a secure connection with the server by default. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies queued file transfers should attempt to immediately establish secure TLS connection with the server by default. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies queued file transfers should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies queued file transfers should use the Secure Shell (SSH) protocol to establish the connection by default. This option will automatically be selected if the connection is established using port 22, the default port for SSH.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

If the ***nMaxFiles*** parameter is INFINITE, memory will be dynamically allocated on the process heap with no limit to the number of files which can be queued. If there is a logic error which causes the application to recursively add files to the queue, or repeatedly queue the same file, this can result in virtual memory being exhausted for the process. You can avoid this by specifying a reasonable maximum queue size, which will cause the **AddFile** method to fail if that limit is exceeded.

When called without any parameters, the constructor will create a queue with no file limit, default options and a default timeout period. The memory allocated for the queue will be released when the class destructor is invoked.

The constructor calls the **FtpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

~CFtpQueue, IsInitialized, CFtpClient::Connect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::~CFtpQueue

```
~CFtpQueue();
```

The **CFtpQueue** destructor releases resources allocated by the current instance of the **CFtpQueue** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CFtpQueue** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::AddFile Method

```
DWORD AddFile(
   DWORD dwQueueMode,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nFileType,
   UINT nTimeout
);

DWORD AddFile(
   DWORD dwQueueMode,
   LPCTSTR lpszHostName,
   UINT nHostPort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nFileType,
   UINT nTimeout
);
```

The **AddFile** method adds a new file to the queue with additional transfer options.

Parameters

dwQueueMode

An integer value which specifies if the file should be downloaded from the server or uploaded to the server. This parameter must be one of the following options and cannot have a value of zero.

Constant	Description
FTP_QUEUE_DOWNLOAD	The file should be queued for downloading from the server to the local system.
FTP_QUEUE_UPLOAD	The file should be queued for uploading from the local system to the server.

lpszHostName

The name of the server to establish a connection with. This may be a fully-qualified domain name or an IP address. If this parameter is NULL or points to an empty string, the ***lpszRemoteFile*** parameter must specify a complete URL which identifies the server as well as the path to the remote file.

nHostPort

The port number which should be used when establishing the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If this parameter is zero and a URL is specified with the ***lpszRemoteFile*** parameter, the port number included in the URL will be used as the default value.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the connection. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that

anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

A null-terminated string which specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

lpszLocalFile

A null-terminated string which specifies the name of the local file to be queued for transfer. This parameter cannot be NULL or point to an empty string. If the file is being queued for an upload, the file must exist on the local system or the method will fail. If the file is being downloaded, it will be created or replaced on the local system.

lpszRemoteFile

A null-terminated string which specifies the name of the file on the server. If the file is being downloaded, the file must exist on the server or the transfer will fail. If the file is being uploaded, the file will be created or replaced on the server. This parameter may specify a complete URL. Note that any values you specify as arguments to this method will override the values specified in the URL. If you want to use a URL as the remote file name, the ***lpszHostName*** parameter should be NULL and the ***nHostPort*** parameter should be zero.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator and can be any of the options used with the **FtpClient::Connect** method. If this value is zero, the default options for the queue will be used. The most common options are:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the file transfer should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the file transfer should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the file transfer should attempt to establish a secure connection with the server by default. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the file transfer should attempt to

	immediately establish secure TLS connection with the server by default. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the file transfer should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the file transfer should use the Secure Shell (SSH) protocol to establish the connection by default. This option will automatically be selected if the connection is established using port 22, the default port for SSH.

nFileType

An integer value which specifies the type of file being transferred. It can be one of the following values:

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **RegisterFileType** method.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

nTimeout

The number of seconds to wait for a response from the server before failing the file transfer. If this value is zero, the default timeout period for the queue will be used.

lpvReserved

A reserved parameter which should always be NULL.

Return Value

If the method succeeds, the return value is a unique identifier which corresponds to the queued file. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will add a file to the current transfer queue. If successful, the value returned is a unique integer which identifies that file in the queue. To obtain information about the queued file, you can call the **GetFile** method and provide this return value as the ***dwFileId*** parameter. To remove a file from the queue, use the **RemoveFile** method.

Files can only be added to the queue when in an idle or paused state. If you attempt to add a file to the queue while the queue manager is transferring files, the method will fail. To determine the current state of the queue, call the **GetStatus** method

There are no fixed limits to the number of files which can be queued for transfer. To release the memory allocated for the queue, call the **Clear** method. To determine the current status of the queue, including how many file transfers have been queued and how many are pending completion, call the **GetStatus** method.

If the local file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

This method will not immediately begin the process of uploading or downloading the file. It will only add the transfer request to the current queue. The file transfers are managed by a background worker thread which is created when the **Start** method is called.

Example

```
// Create a new queue
CFtpQueue pQueue = new CFtpQueue();

if (!pQueue->IsInitialized())
    return;

// Add a file to the queue
DWORD dwFileId = pQueue->AddFile(
                     FTP_QUEUE_DOWNLOAD,
                     lpszHostName,
                     FTP_PORT_DEFAULT,
                     lpszUserName,
                     lpszPassword,
                     lpszLocalFile,
                     lpszRemoteFile);

// Display information about the queued file
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;
```

```
    if (pQueue->GetFile(dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (pQueue->Start())
{
    pQueue->Wait(INFINITE);
    pQueue->Stop();
}

// Remove all files from the queue
delete pQueue;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CFtpQueue, ~CFtpQueue, FindFile, GetFile, GetFirstFile, GetNextFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::AttachHandle Method

```
VOID AttachHandle(
   HQUEUE hQueue
);
```

The **AttachHandle** method attaches the specified queue handle to the current instance of the class.

Parameters

hQueue

A handle to the queue.

Return Value

None.

Remarks

This method is used to attach a queue handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that queue.

If a handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous queue from being destroyed, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hQueue*** parameter is presumed to be a valid queue handle and no checks are performed to ensure that the handle is valid. Specifying an invalid queue handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Cancel Method

```
BOOL Cancel(
   DWORD dwFileId
);
```

The **Cancel** method cancels the specified file transfer.

Parameters

dwFileId

An optional unsigned integer value which uniquely identifies the file in the queue. If this parameter is omitted or the value is zero, the method will cancel the current file transfer in progress. If queue transfers have been suspended, this parameter cannot be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

When this method is called, the queued file transfer may not immediately stop. An internal flag is set which causes the file transfer to exit with an error and the queue manager will begin processing the next file in the queue. If the queue is in an idle state, this method will fail.

It is permitted to call the **Cancel** method from within a queue event handler to cancel the current file transfer. In this case, the *dwFileId* parameter should be zero. If you specify a file which has not been transferred, it will be flagged as canceled and skipped by the queue manager when processing the queue. If you specify a file which has already been processed, this method will fail.

A canceled file transfer is also considered a failed transfer. When you call **GetStatus** after a queued file transfer is canceled, the *dwFailedFiles* member of the FTPQUEUESTATUS structure will be incremented to reflect this change. You can determine the status of an individual file transfer by calling the **GetFile** method and checking the value of the *dwQueueFlags* member of the FTPQUEUEDFILE structure.

The **Reset** method can be used to reset the state of previously canceled transfers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Cancel, GetFile, GetStatus, Reset, Resume, Start, Suspend, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Clear Method

```
BOOL Clear();
```

The **Clear** method removes all files from the queue.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

This method can only be called when the queue is in an idle state. An error will be returned if the method is called while the queue manager is paused or actively transferring files in the queue. To determine the current state of the queue, call the **GetStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Cancel, GetStatus, Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::DetachHandle Method

```
HQUEUE DetachHandle();
```

The **DetachHandle** method detaches the queue handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the queue handle associated with the current instance of the class object. If an error occurs, the value INVALID_QUEUE will be returned.

Remarks

This method is used to detach a queue handle created by the class for use with the SocketTools API. Once the handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, the control is attempting to connect to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::EnumFiles Method

```
LONG EnumFiles(
   LPCTSTR lpszFileMask,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPDWORD lpFileList,
   LONG nMaxFiles
);
```

The **EnumFiles** method returns a list of files in the current transfer queue.

Parameters

lpszFileMask

A null-terminated string which specifies a file name which can include wildcards. Only those files which match this value will be enumerated. The character '?' will match against any single character, and '*' will match any number of characters. If this parameter is NULL or points to an empty string, all eligible files in the queue will be matched.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. Only those files which match the queue mode will be enumerated. This is a bitmask which may be one or more of the following values:

Constant	Description
FTP_QUEUE_ALL	Match the file name to all files in the queue.
FTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
FTP_QUEUE_UPLOAD	Match file names which are queued for upload.
FTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the enumeration. If this value is zero, no files will be excluded; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_QUEUE_FLAG_NONE	No files should be excluded.
FTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
FTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
FTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being

	uploaded or downloaded.

lpFileList

A pointer to an array of unsigned integer values which will contain the unique file identifiers for each matching file in the queue. This parameter must specify an array large enough to store all of the file identifiers, otherwise the method will fail with a `ST_ERROR_BUFFER_TOO_SMALL` error. If this parameter is NULL, the method will return the number of matching files.

nMaxFiles

An integer value which specifies the maximum number of file identifiers which can be copied into the *lpFileList* array. If the *lpFileList* parameter is not NULL, this value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of matching files. If the queue is empty or there are no matching files, this method will return zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method populates an array of unique file identifiers which can be used to obtain information about the files in the current queue. The values returned in the *lpFileList* array can be used in conjunction with the **GetFile** method. This provides an alternative to using the **GetFirstFile** and **GetNextFile** methods.

The application should not make any assumptions about the value of the file identifiers returned by this method. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

Example

```
// Get the total number of files in the queue
LONG nFiles = pQueue->GetCount();

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];
    nFiles = pQueue->EnumFiles(NULL, FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE,
pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
        FTPQUEUEDFILE queuedFile;

        if (pQueue->GetFile(pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;
```

```
        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, FindFile, GetFile, GetFirstFile, GetNextFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::FindFile Method

```
DWORD FindFile(
   LPCTSTR lpszFileName,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles
);
```

The **FindFile** method finds a file in the transfer queue which matches the search criteria.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file to search for. The string may contain the wildcard character '?' to match against any single character, and '*' will match any number of characters. This parameter cannot be NULL or point to an empty string.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. This is a bitmask which may be one or more of the following values:

Constant	Description
FTP_QUEUE_ALL	Match the file name to all files in the queue.
FTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
FTP_QUEUE_UPLOAD	Match file names which are queued for upload.
FTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_QUEUE_FLAG_NONE	No files should be excluded from the search.
FTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
FTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
FTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being uploaded or downloaded.

Return Value

If the method succeeds, the return value is a unique identifier for the file. To obtain information about the file, call the **GetFile** method. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns a unique file identifier for the first file which matches the specified file name in the queue. If the *lpszFileName* parameter includes wildcard characters, this method will return the first file in the queue which matches the name. File name matches are not case-sensitive, even when matching against remote file names on servers which use case-sensitive naming, such as UNIX based servers.

To perform more complex searches based on the file name, or to find multiple files, you can use the **GetFirstFile** and **GetNextFile** methods to iterate through all queued files.

Example

```
// Find a local file in the download queue
DWORD dwFileId = pQueue->FindFile(lpszFileName,
                                  FTP_QUEUE_DOWNLOAD | FTP_QUEUE_LOCAL,
                                  FTP_QUEUE_FLAG_NONE);

// Display information about the queued file
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;

    if (pQueue->GetFile(dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, EnumFiles, GetFile, GetFirstFile, GetNextFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetClient Method

```
HCLIENT GetClient();
```

The **GetClient** method returns a handle to the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value a handle to a client session. If the method fails, it will return INVALID_CLIENT. To get extended error information, call **GetLastError**.

Remarks

The **GetClient** method returns the handle to a client session which may be used with other methods. A valid handle will only be returned if the queue manager is currently uploading or downloading a file. If the queue is active but there are no file transfers in progress at the time this method is called, it will fail and set the last error code to `ST_ERROR_NO_QUEUED_TRANSFER`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpQueue, ~CFtpQueue, GetTransferStatus, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetCount Method

```
LONG GetCount();
```

The **GetCount** method returns the total number of files which have been queued for transfer.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of files in the queue. If the method fails, it will return FTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpQueue, ~CFtpQueue, EnumFiles, GetStatus, GetFile, GetFirstFile, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetFile Method

```
BOOL GetFile(
   LPCTSTR lpszFileName,
   LPFTPQUEUEDFILE lpFileInfo
);

BOOL GetFile(
   DWORD dwFileId,
   LPFTPQUEUEDFILE lpFileInfo
);
```

The **GetFile** method returns information about the specified file in the transfer queue.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file in the queue. The name should always be the name of the remote file to search for.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

lpFileInfo

A pointer to a FTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the file identifier is not valid or the file has been removed from the queue, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The file identifier can be obtained by several different methods, including **EnumFiles** and **GetStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

If you call the version of this method which accepts a file name, it will use the **FindFile** method to obtain the file ID for the first file name which matches a remote file in the queue. This approach is only recommended if you know that the queued file names will be unique.

Example

```
// Get the number of files in the queue
LONG nFiles = pQueue->GetCount();

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];

    nFiles = pQueue->EnumFiles(pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
```

```
        FTPQUEUEDFILE queuedFile;

        if (pQueue->GetFile(pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumFiles, GetStatus, GetFile, GetFirstFile, GetNextFile, GetStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetFirstFile Method

```
DWORD GetFirstFile(
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPFTPQUEUEDFILE lpFileInfo
);
```

The **FtpGetFirstFile** method returns information about the first file in the transfer queue.

Parameters

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. It may be one of the following values:

Constant	Description
FTP_QUEUE_ALL	Return information about all files in the queue.
FTP_QUEUE_DOWNLOAD	Return information for files which are queued for download.
FTP_QUEUE_UPLOAD	Return information for files which are queued for upload.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_QUEUE_FLAG_NONE	No files should be excluded.
FTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
FTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
FTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being uploaded or downloaded.

lpFileInfo

A pointer to a FTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no queued files which match the specified parameters, or

the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns information about the first file in the current transfer queue. It is used in conjunction with the **GetNextFile** method to obtain information about all queued files.

Example

```
// List all files in a file transfer queue
FTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = pQueue->GetFirstFile(FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE, &queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case FTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case FTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
    }

    dwFileId = pQueue->GetNextFile(&queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, EnumFiles, FindFile, GetFile, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetHandle Method

```
HQUEUE GetHandle();
```

The **GetHandle** method returns the queue handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the queue handle associated with the current instance of the class object. If an error occurs, the value INVALID_QUEUE will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetNextFile Method

```
DWORD GetNextFile(
   LPFTPQUEUEDFILE lpFileInfo
);
```

The **GetNextFile** method returns information about the next file in the transfer queue.

Parameters

lpFileInfo

A pointer to a FTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no additional queued files which match the exclusion criteria specified with the call to **GetFirstFile**, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns information about the next file in the current transfer queue after an initial call to **GetFirstFile**. To obtain a list of all matching files in the queue, call this method repeatedly until it returns a value of zero. When information about the last file in the queue has been returned, the last error code will be set to `ST_ERROR_END_OF_QUEUE`.

Example

```
// List all files in a file transfer queue
FTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = pQueue->GetFirstFile(FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE, &queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case FTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case FTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
    }

    dwFileId = pQueue->GetNextFile(&queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, EnumFiles, FindFile, GetFile, GetFirstFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetStatus Method

```
INT GetStatus(
   LPFTPQUEUESTATUS lpQueueStatus
);
```

The **GetStatus** method returns information about the specified file in the transfer queue.

Parameters

lpQueueStatus

A pointer to a FTPQUEUESTATUS structure which will contain information about the current state of the file transfer queue. If this parameter is NULL, the method will ignore the parameter and only return the current status of the queue.

Return Value

If the method succeeds, the return value is the current queue status. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**. The following status values may be returned:

Value	Description
FTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before **FtpStartQueue** has been called or after **FtpStopQueue** has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
FTP_QUEUE_STATUS_ACTIVE	The queue manager is active and files are currently being uploaded or downloaded.
FTP_QUEUE_STATUS_PAUSED	The queue manager is active although file transfers are currently paused. The queue enters this state after the **FtpSuspendQueue** method is called and resumes file transfers after the **FtpResumeQueue** method is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpQueue, ~CFtpQueue, Cancel, Reset, Resume, Start, Suspend, Stop, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::GetTransferStatus Method

```
INT GetTransferStatus(
   LPFTPTRANSFERSTATUS lpStatus
);

INT GetTransferStatus(
   LPFTPTRANSFERSTATUSEX lpStatus
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an FTPTRANSFERSTATUS or FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. This method would typically be called from within a callback function in response to an FTP_EVENT_PROGRESS event notification.

The *dwBytesTotal* and *dwBytesCopied* members of the **FTPTRANSFERSTATUSEX** structure are declared as unsigned 64-bit integers rather than 32-bit integers. To obtain accurate file transfer information, this extended version of the structure should be used with files that are larger than 4GiB.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CFtpQueue, ~CFtpQueue, FtpEventProc, GetClient, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::IsEmpty Method

```
BOOL IsEmpty();
```

The **IsEmpty** method determines if the current queue is empty.

Parameters

None.

Return Value

If no files have been added to the queue, the return value is non-zero. If one or more files have been added to the queue, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AddFile Clear GetFile, IsIdle, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::IsIdle Method

```
BOOL IsIdle();
```

The **IsIdle** method determines if the queue is idle.

Parameters

None.

Return Value

If the queue is idle and no files are being transferred, this method returns a non-zero value. If the queue manager is currently transferring files, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AddFile Cancel, Clear IsEmpty, Reset, Resume, Start, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpQueue, ~CFtpQueue, Clear GetClient, GetHandle Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::RegisterFileType Method

```
BOOL RegisterFileType(
   LPCTSTR lpszExtension,
   UINT nFileType
);
```

The **RegisterFileType** method associates a file name extension with a specific file type.

Parameters

lpszExtension

A null-terminated string which specifies the file name extension. If this parameter is NULL or points to an empty string, the default file type will be changed for the client session.

nFileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values.

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary image and no data conversion of any type is performed on the file. This is typically the default file type for data file transfers. If the type of file that is being transferred is unknown, this file type should always be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **RegisterFileType** method is used to associate specific file types with file name extensions. The class has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that global list. Registered file types are shared by all queues created by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, FindFile, GetFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::RemoveFile Method

```
BOOL RemoveFile(
   DWORD dwFileId
);

BOOL RemoveFile(
   LPCTSTR lpszFileName
);
```

The **RemoveFile** method removes the specified file from the queue.

Parameters

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

lpszFileName

A null-terminated string which specifies the name of the file to remove. If this version of the method is called, this parameter cannot be NULL or point to an empty string. The file name may include wildcard characters if you wish to remove multiple files. The character '?' will match against any single character, and '*' will match any number of characters.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

Files can only be removed from the queue when in an idle or paused state. If you attempt to remove a file while the queue manager is in the process of uploading or download files, the method will fail.

The file identifier can be obtained by several different methods, including **EnumFiles** and **GetStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

If you call this method using a file name, the **EnumFiles** method will be used to enumerate all remote file names which match and remove them from the queue. File name matches are not case-sensitive, even when matching against file names on servers which use case-sensitive naming, such as UNIX based servers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AddFile Clear IsEmpty, Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Reset Method

```
BOOL Reset(
   DWORD dwOptions
);
```

The **Reset** method resets the state of all file transfers in the current queue.

Parameters

dwOptions

An unsigned integer value which specifies which files should be reset in the queue. It may be one of the following values:

Value	Description
FTP_QUEUE_RESET_ALL	All files in the queue should be reset to their initial state.
FTP_QUEUE_RESET_COMPLETED	All files in the queue which have been successfully transferred will be reset to their initial state.
FTP_QUEUE_RESET_FAILED	All files in the queue which were not transferred will be reset to their initial state.
FTP_QUEUE_RESET_CANCELED	All files in the queue which were canceled will be reset to their initial state.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Start** method is called multiple times, it will not attempt to transfer files which have already been successfully copied, and it will not attempt to re-transfer files if the previous transfer failed. This method resets queued files back to their initial state, prior to when the queue manager attempted to perform the transfer. The most common use would be to call the method with the FTP_QUEUE_RESET_FAILED mode and then call **Start** to retry failed or canceled file transfers.

Because canceled file transfers are also considered failed transfers, specifying FTP_QUEUE_RESET_FAILED will reset queued files which either encountered an error during the transfer or were explicitly canceled by calling the **Cancel** method. The FTP_QUEUE_RESET_CANCELED option will only reset the state of queued file transfers which were canceled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

AddFile Cancel, Clear, IsEmpty, Resume, Start, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Resume Method

```
BOOL Resume();
```

The **Resume** method resumes transfers in the queue.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

This method signals the queue manager to resume processing files in the transfer queue after the **Suspend** method has been called. This method will fail if the queue is in an idle (stopped) state, and will be ignored if the queue manager is already transferring files. The **GetStatus** method can be called to obtain the current status of the queue.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Cancel, Clear, GetStatus, Start, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Failure is typically indicated by a return value such as FALSE, NULL, INVALID_QUEUE or FTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by calling the **GetLastError** method. The use of **GetLastError** is optional; an application can use the method to determine the specific reason for a failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** function in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Start Method

```
BOOL Start(
   DWORD dwQueueMode,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **Start** method begins transferring files in the queue.

Parameters

dwQueueMode

An unsigned integer which specifies which files in the queue should be transferred. It may be one of the following values:

Constant	Description
FTP_QUEUE_ALL	All files in the queue should be transferred.
FTP_QUEUE_DOWNLOAD	Transfer only those files which have been queued for download.
FTP_QUEUE_UPLOAD	Transfer only those files which have been queued for upload.

lpEventProc

Specifies the procedure-instance address of an application defined callback function. For more information about the callback method, see the description of the **FtpEventProc** callback method. If this parameter is NULL, there will be no callback notifications.

dwParam

A user-defined integer value that is passed to the callback method. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

Queued file transfers are performed asynchronously using a background worker thread. If you provide the address to an event callback function, that method will always be invoked in the context of the queue manager thread. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself and you should avoid calling any methods which could cause the thread to block. For example, you should not attempt to establish other network connections from within the event handler.

If your application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback method. Controls should only be modified by the same UI thread that created their window. One common approach to address this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the ***this*** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

The **Wait** method can be used to wait for the queue manager to complete processing the transfer queue.

Example

```
// Create a new queue
CFtpQueue *pQueue = new CFtpQueue();

// Add a file to the queue
pQueue->AddFile(lpszLocalFile, lpszRemoteFile);

// Start the queued file transfers and wait for it to complete
if (pQueue->Start(FTP_QUEUE_ALL))
{
    pQueue->Wait(INFINITE);
    pQueue->Stop();
}

// Remove all files from the queue
delete pQueue;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpQueue, ~CFtpQueue, Cancel, Clear, GetStatus, Resume, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Stop Method

```
BOOL Stop();
```

The **Stop** method stops transferring queued files.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

When the **Stop** method is called and there is a file transfer in progress, it will not immediately stop the upload or download of the file. Instead, the queue manager is signaled to stop processing additional files in the queue after the transfer has completed. To wait for the current transfer to complete, call the **Wait** method.

It is permitted to call the **Stop** method from within a queue event handler. In this case, the current file transfer will complete and the queue manager will terminate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Cancel, Clear, GetStatus, Resume, Start, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Suspend Method

```
BOOL Suspend();
```

The **Suspend** method pauses all file transfers for the queue.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

If there is a file transfer in progress when this method is called, the transfer will complete normally. The queue manager will enter a suspended state after the transfer has completed and before it begins processing the next file in the queue.

It is permitted to call the **Suspend** method from within a queue event handler. In this case, the current file transfer will complete and the queue manager will stop processing additional files in the queue until it is resumed or stopped.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Cancel, Clear, GetStatus, Resume, Start, Stop, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpQueue::Wait Method

```
BOOL WINAPI FtpWaitForQueue(
   DWORD dwMilliseconds,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);

BOOL WINAPI FtpWaitForQueue(
   DWORD dwMilliseconds
);
```

The **Wait** method waits for the specified queue to complete the file transfers.

Parameters

dwMilliseconds

An unsigned integer value which specifies the number of milliseconds to wait for the queue to complete processing. If this value is zero, the method will return immediately. If this value is INFINITE (0xFFFFFFFF), the method will block indefinitely until all files in the queue have been processed.

lpdwElapsed

A pointer to an unsigned integer which will contain the number of milliseconds the queue manager has been active when the method returns. If this information is not needed, this parameter can omitted or it can be NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

lpdwError

A pointer to an unsigned integer which will contain the last error code set if a file transfer has failed or was canceled. If this information is not needed, this parameter can be omitted or NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

Return Value

If the method succeeds, the return value is non-zero, which means the queue manager has completed transferring all queued files. If the method fails, or the timeout period elapses, it will return zero. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until either the queue manager has completed processing all files in the queue, or until the specified number of milliseconds have elapsed. It is important to note that Windows messages will not be processed during this time. If you call this method within the main UI thread, it can potentially cause the application to become non-responsive. To determine the current state of the queue without blocking the current thread, call the **GetStatus** method.

If the *dwMilliseconds* parameter is non-zero and file transfers have not completed within the specified amount of time, the method will return zero and the last error code will be set to `ST_ERROR_OPERATION_TIMEOUT`.

If an event callback function has been specified when starting the queue, you should never call this method within the event handler. Attempting to do so can potentially create a situation which will cause the application to become non-responsive. The method will return zero if it determines it's being called within the context of the queue manager thread and will set the last error code to

ST_ERROR_THREAD_DEADLOCK.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

Cancel, Clear, GetStatus, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Queue Data Structures

- FTPQUEUEDFILE
- FTPQUEUESTATUS
- FTPTRANSFERSTATUS
- FTPTRANSFERSTATUSEX
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPQUEUEDFILE

This structure is used by the **GetFile**, **GetFirstFile** and **GetNextFile** methods to return information about a file in the transfer queue.

```
typedef struct _FTPQUEUEDFILE
{
    DWORD          dwFileId;
    DWORD          dwQueueMode;
    DWORD          dwQueueFlags;
    DWORD          dwTimeElapsed;
    DWORD          dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
    TCHAR          szRemoteFile[FTP_MAXURLPATHLEN];
} FTPQUEUEDFILE, *LPFTPQUEUEDFILE;
```

Members

dwFileId

An unsigned integer which specifies a unique identifier for the queued file. The application should not make any assumptions about the value of the file identifiers. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

dwQueueMode

An unsigned integer value which specifies how the file was queued for transfer. It may be one of the following values:

Constant	Description
FTP_QUEUE_DOWNLOAD	The file was queued for download.
FTP_QUEUE_UPLOAD	The file was queued for upload.

dwQueueFlags

An unsigned integer which specifies one or more bit flags which provides information about the status of the file transfer. It may be one or more of the following values:

Constant	Description
FTP_QUEUE_FLAG_NONE	The file is pending transfer in queue.
FTP_QUEUE_FLAG_COMPLETED	The file has been transferred successfully. If this flag is set, no errors were encountered during the upload or download.
FTP_QUEUE_FLAG_FAILED	The file transfer failed. If this flag is set, the *dwError* member of this structure will contain the error code associated with the failed transfer.
FTP_QUEUE_FLAG_CANCELED	The file transfer was canceled. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being uploaded or downloaded.

dwTimeElapsed

An unsigned integer which specifies the number of milliseconds required to complete the transfer. This value will be zero unless the file has been transferred successfully.

dwLastError

An unsigned integer which specifies the last error code for a failed transfer. If the file was transferred successfully, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the file transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

szLocalFile

A null-terminated string which specifies the full path to the local file being transferred.

szRemoteUrl

A null-terminated string which specifies the complete URL to the remote file being transferred.

Remarks

It is possible that the ***szLocalFile*** and ***szRemoteFile*** structure members will be different than the values passed to the **AddFile** method. Those values are normalized, with any relative paths converted to absolute paths. Internationalized domain names will be encoded and the URL paths will be collapsed, removing any extraneous path information. For example, if the remote file name is specified as `ftp://ftp.server.tld/folder1/../folder2/filename.txt` it would be normalized as `ftp://ftp.server.tld/folder2/filename.txt`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, GetFile, GetFirstFile, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPQUEUESTATUS

This structure is used by the **GetStatus** method to return information about the current status of the file transfer queue.

```
typedef struct _FTPQUEUESTATUS
{
    DWORD dwStatus;
    DWORD dwThreadId;
    DWORD dwQueuedFiles;
    DWORD dwPendingFiles;
    DWORD dwCopiedFiles;
    DWORD dwFailedFiles;
    DWORD dwTimeElapsed;
    DWORD dwCurrentFile;
    DWORD dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    ULARGE_INTEGER uiBytesTotal;
} FTPQUEUESTATUS, *LPFTPQUEUESTATUS;
```

Members

dwStatus

An unsigned integer which specifies the current status of the queue. It can be one of the following values:

Value	Description
FTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before the **Start** method has been called or after the **Stop** method has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
FTP_QUEUE_STATUS_ACTIVE	Files in the queue are being uploaded or downloaded. The ***dwCurrentFile*** member of this structure identifies the file which is currently being transferred.
FTP_QUEUE_STATUS_PAUSED	File transfers are currently paused. The queue enters this state after the **Suspend** method is called and resumes file transfers after the **Resume** method is called.

dwThreadId

An unsigned integer which specifies the thread ID for the queue manager which is performing the file transfers. All file transfers are performed asynchronously in a background worker thread. If this structure member is zero, the queue is idle and not performing file transfers. The application should not use the thread ID to obtain a handle to suspend or terminate the thread. This can potentially result in unexpected behavior or instability within the application.

dwQueuedFiles

An unsigned integer which specifies the total number of queued files. This value includes pending and completed file transfers.

dwPendingFiles

An unsigned integer which specifies the number of files which are queued to be transferred.

dwCopiedFiles

An unsigned integer which specifies the number of files which have been successfully transferred. This value reflects the total number of files which have the FTP_QUEUE_FLAG_COMPLETED status.

dwFailedFiles

An unsigned integer which specifies the number of files which have failed to transfer or the file transfer has been canceled. This value reflects the total number of files which have the FTP_QUEUE_FLAG_FAILED status.

dwTimeElapsed

An unsigned integer which specifies the total amount of time, in milliseconds, the queue manager has been active performing a file transfer. If the queue is idle, this value will reflect the total run time for the previously active queue. A value of zero indicates the queue was never active or the queue state has been reset with a call to the **Reset** method.

dwCurrentFile

An unsigned integer which specifies the unique identifier for the current file being transferred. If no file transfer is in progress, this member will have a value of zero. If the value is non-zero, it can be passed to the **GetFile** method to obtain information about the queued file.

dwLastError

An unsigned integer which specifies the last error code for a failed file transfer. If there have been no errors, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the current or last transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

uiBytesTotal

A value which specifies the total number of bytes which were copied during the current queue run. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

GetFile, GetStatus, Cancel, Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUS

This structure is used by the **GetTransferStatus** method to return information about a file transfer in progress.

```
typedef struct _FTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
   TCHAR   szLocalFile[FTP_MAXFILENAMELEN];
   TCHAR   szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUS, *LPFTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

szLocalFile

A null-terminated string which specifies the local file that is being copied to or from the server.

szRemoteFile

A null-terminated string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FTPTRANSFERSTATUSEX** structure which uses 64-bit integers for the file size.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

GetTransferStatus, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUSEX

This structure is used by the **GetTransferStatus** method to return information about a file transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPTRANSFERSTATUSEX
{
   ULARGE_INTEGER dwBytesTotal;
   ULARGE_INTEGER dwBytesCopied;
   DWORD          dwBytesPerSecond;
   DWORD          dwTimeElapsed;
   DWORD          dwTimeEstimated;
   DWORD          dwReserved;
   TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
   TCHAR          szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUSEX, *LPFTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

dwReserved

This structure member is reserved for future use.

szLocalFile

A null-terminated string which specifies the local file that is being copied to or from the server.

szRemoteFile

A null-terminated string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h

See Also

GetTransferStatus, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Server Class

Implements a server that enables the application to send and receive files using the File Transfer Protocol.

Reference

- Data Members
- Class Methods
- Event Handlers
- Data Structures
- Error Codes

Library Information

File Name	CSFTSV12.DLL
Version	12.0.3024.1286
LibID	BB27183D-4826-4F44-9EF7-128EADE765D4
Import Library	CSFTSV12.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

This class provides an interface for implementing an embedded, lightweight server that can be used to exchange files with a client using the standard File Transfer Protocol. The server can accept connections from any third-party application or a program developed using the SocketTools FTP client API.

The application specifies an initial server configuration and then responds to events that are raised by the API when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard FTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server supports active and passive mode file transfers, has compatibility options for NAT router and firewall support, and provides support for secure file transfers using explicit TLS. Secure connections require a valid security certificate to be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer Public Data Members

Member Variables	Description
m_dwOptions	Options specified when creating an instance of the server
m_nAuthFail	The maximum number of user authentication failures permitted per client session
m_nAuthTime	The maximum number of seconds a client has to successfully authenticate the session
m_nExecTime	The maximum number of seconds that the server will permit an external command to execute
m_nIdleTime	The maximum number of seconds a client can be idle before the server terminates the session
m_nLogFormat	The format used by the server to log client activity
m_nLogLevel	The level of detail included in the server log file
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_nMaxGuests	The maximum number of anonymous client sessions accepted by the server
m_nMaxPort	The maximum port number used by the server for passive data connections
m_nMinPort	The minimum port number used by the server for passive data connections

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_dwOptions

```
DWORD m_dwOptions;
```

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see Server Option Constants. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer, GetOptions, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nAuthFail

```
UINT m_nAuthFail;
```

The maximum number of user authentication failures permitted per client session.

Remarks

The **m_nAuthFail** data member is a public variable that specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies that the default configuration limit of 3 authentication attempts per login should be allowed. The maximum number of authentication attempts is 10. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nAuthTime

```
UINT m_nAuthTime;
```

The maximum number of seconds a client has to successfully authenticate the session.

Remarks

The **m_nAuthTime** data member is a public variable that specifies the maximum number of seconds that a client session must authenticate itself. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 20 seconds and the maximum value is 300 seconds (5 minutes). This value is used to ensure that a client has successfully authenticated itself within a limited period of time.

This time limit prevents a potential denial-of-service attack against the server where clients establish connections and hold them open without authentication. In conjunction with the **m_nAuthFail** data member, this also limits the ability of a client to attempt to probe the server for valid username and password combinations. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nExecTime

```
UINT m_nExecTime;
```

The maximum number of seconds that the server will permit an external command to execute.

Remarks

The **m_nExecTime** data member is a public variable that specifies the maximum number of seconds that an external program is permitted to run on the server. External programs are registered using the **RegisterProgram** method, and are executed by the client sending the SITE EXEC command to the server. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nIdleTime

```
UINT m_nIdleTime;
```

The maximum number of seconds a client can be idle before the server terminates the session.

Remarks

The **m_nIdleTime** data member is a public variable that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 900 seconds (15 minutes). If the value is non-zero, the minimum value is 60 seconds and the maximum value is 7200 seconds (2 hours). This value is used to initialize the default idle timeout period for each client session. A client may request that the server change the idle timeout period for its session by sending the SITE IDLE command. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nLogFormat

```
UINT m_nLogFormat;
```

The format used when updating the server log file.

Remarks

The **m_nLogFormat** data member is a public variable that specifies the format of the log file that is created or updated by the server. It may be one of the following values:

Value	Description
FTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

CFtpServer, GetLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nLogLevel

```
UINT m_nLogLevel;
```

The level of detail included in the server log file.

Remarks

The **m_nLogLevel** data member is a public variable that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If the **m_nLogFormat** data member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format.

The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

CFtpServer, GetLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nMaxClients

```
UINT m_nMaxClients;
```

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

CFtpServer, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nMaxClientsPerAddress

```
UINT m_nMaxClientsPerAddress;
```

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

See Also

CFtpServer, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nMaxGuests

```
UINT m_nMaxGuests;
```

The maximum number of anonymous users that are permitted to connect to the server.

Remarks

The **m_nMaxGuests** data member is a public variable that specifies the maximum number of anonymous users that are permitted to establish a connection with the server. After this limit is reached, the server will send an error response indicating that the server is unavailable, and then immediately terminate the session. Unlike the limit on the total number of client connections, this limit is only checked after the client has requested authentication and has logged in as an anonymous user. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nMaxPort

```
UINT m_nMaxPort;
```

The maximum port number used by the server for passive data connections.

Remarks

The **m_nMaxPort** data member is a public variable that specifies the maximum range of port numbers that will be used with passive data connections. A value of zero specifies the default value of 65535 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be greater than the value of the **m_nMinPort** data member. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::m_nMinPort

```
UINT m_nMinPort;
```

The minimum port number used by the server for passive data connections.

Remarks

The **m_nMinPort** data member is a public variable that specifies the minimum range of port numbers that will be used with passive data connections. A value of zero specifies that the default value of 30000 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be less than the value of the **m_nMaxPort** data member. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer Methods

Class	Description
CFtpServer	Constructor which initializes the current instance of the class
~CFtpServer	Destructor which releases resources allocated by the class
Method	Description
AddVirtualUser	Add a new virtual user for the specified server
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
AuthenticateClient	Authenticate the client and assign access rights for the session
ChangeClientDirectory	Change the current working directory for the client session
DeleteVirtualUser	Delete a virtual user from the specified server
DetachHandle	Detach the server handle from the current instance of this class
DisableCommand	Disable a specific server command
DisableTrace	Disable the logging of network function calls
DisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
EnableClientAccess	Enable or disable access rights for the specified client session
EnableCommand	Enable a specific server command
EnableTrace	Enable logging of network function calls to a file
EnumClients	Returns a list of active client connections established with the server
GetActiveClient	Return the client ID for the active client session associated with the current thread
GetAddress	Return the IP address for the server
GetClientAccess	Return the access rights that have been granted to the client session
GetClientAddress	Return the IP address of the specified client session
GetClientCredentials	Return the credentials for the specified client session
GetClientDirectory	Return the current working directory for a client session
GetClientFileType	Return the current file type used for transfers by the specified client
GetClientHomeDirectory	Return the home directory for an authenticated client session
GetClientIdentity	Return the identity of the specified client session
GetClientIdleTime	Return the idle timeout period for the specified client
GetClientLocalPath	Return the full local path for the specified virtual path
GetClientServer	Return the handle to the server that created the specified client session
GetClientThreadId	Returns the thread ID associated with the specified client session

GetClientUserName	Return the user name associated with the specified client session
GetClientVirtualPath	Return the virtual path for a local file on the server
GetCommandFile	Return the full path to the local file name or directory specified by the client
GetCommandLine	Return the complete command line issued by the client
GetCommandName	Return the name of the last command issued by the client
GetCommandParam	Return the value of the specified parameter for the command issued by the client
GetCommandParamCount	Return the number of parameters to the current command issued by the client
GetCommandResult	Return the result code and a description of the last command processed by the server
GetCommandUsage	Return the number of times a specific command has been issued by all clients
GetDirectory	Return the full path to the root directory assigned to the specified server
GetHandle	Return the server handle associated with the class instance
GetIdentity	Return the identity and version information for the specified server
GetLastError	Return information about the last server error that occurred
GetLogFile	Return the current log file format and full path for the file
GetMemoryUsage	Return the amount of memory allocated for the server and all client sessions
GetName	Return the host name assigned to the server or specified client session
GetOptions	Return the options specified for this instance of the server
GetPriority	Return the current priority assigned to the specified server
GetProgramExitCode	Return the exit code of the last program executed by the client
GetProgramName	Return the name of the last program executed by the client
GetProgramOutput	Return a copy of the standard output from the last program executed by the client
GetProgramText	Return a copy of the standard output from the last program in a string buffer
GetRenamedFile	Return the original name of a file being renamed by the client
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetTransferInfo	Return information about the current file transfer for the client session
GetUuid	Return the UUID assigned to the specified server
IsActive	Determine if the server has been started
IsClientAnonymous	Determine if the specified client has authenticated as an anonymous user
IsClientAuthenticated	Determine if the specified client session has been authenticated
IsCommandEnabled	Determine if the specified command is currently enabled or disabled
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the server is listening for client connections
PreProcessEvent	Filter server events before being processed by the default event handler

RegisterProgram	Register a program for use with the SITE EXEC command
RenameServerLogFile	Rename or delete the current log file being updated by the server
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SendResponse	Send a result code and optional message to the client in response to a command
SetAddress	Change the IP address that the server will use with passive data connections
SetCertificate	Set the name of the certificate to be used with secure connections.
SetClientAccess	Change the access rights associated with the specified client session
SetClientFileType	Change the current file type used for transfers by the specified client
SetClientIdentity	Change the identity string associated with the specified client session
SetClientIdleTime	Change the idle timeout period for the specified client session
SetCommandFile	Change the name of the local file or directory that is the target of the current command
SetDirectory	Specify the local directory that will be used as the server root directory
SetLastError	Set the last error code for the specified server session
SetLogFile	Change the current log format, level of detail and file name
SetIdentity	Change the identity and version information for the specified server
SetName	Change the hostname assigned to the server or specified client session
SetOptions	Change the options specified for this instance of the server
SetPriority	Change the priority assigned to the specified server
SetStackSize	Change the initial size of the stack allocated for threads created by the server
SetUuid	Assign a UUID to be associated with this instance of the server
Start	Start the server and begin accepting client connections
Stop	Stop the server and terminate all active client connections
Suspend	Suspend accepting client connections on the specified server
Throttle	Limit the number of active client connections, connections per address and connection rate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::~CFtpServer

```
~CFtpServer();
```

The **CFtpServer** destructor releases resources allocated by the current instance of the **CFtpServer** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CFtpServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::AddVirtualUser Method

```
BOOL AddVirtualUser(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   DWORD dwUserAccess,
   LPCTSTR lpszDirectory
);
```

Add a new virtual user for the specified server.

Parameters

lpszUserName

A null-terminated string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the method will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A null-terminated string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the method will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see User Access Constants.

lpszDirectory

A null-terminated string which specifies the directory that will be the virtual user's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created the first time that the virtual user successfully logs in to the server. If this parameter is NULL or an empty string, a default home directory will be created for the virtual user.

Return Value

If the method succeeds, the return value is non-zero. If the username or password contain invalid characters, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualUser** method adds a virtual user that is associated with the specified virtual host. When a client connects with the server and provides authentication credentials, the server will check if the username has been created using this method. If a match is found, the client access rights will be updated.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this method is called with a username that already exists, that record is replaced with the values passed to this method. You cannot use this method to create a virtual user named "anonymous".

The virtual users created by this method exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this method for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::AsyncNotify Method

```
BOOL AsyncNotify(
   HWND hWnd,
   UINT uMsg
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the ***hWnd*** parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the ***hWnd*** parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The ***wParam*** message parameter will contain the notification message and the ***lParam*** message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Value	Description
FTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the **Start** method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The ***lParam*** message parameter will specify the handle to the server.
FTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the **Resume** method to resume accepting

	client connections. The *lParam* message parameter will specify the handle to the server.
FTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the **Suspend** method. This notification message may be sent to the application multiple times over the lifetime of the server. The *lParam* message parameter will specify the handle to the server.
FTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the **Restart** method. Note that the server socket handle provided by the *lParam* message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The *lParam* message parameter will specify the handle to the server.
FTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The *lParam* message parameter will specify the unique ID of the client that connected to the server.
FTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the **Stop** method. The *lParam* message parameter will specify the unique ID of the client that disconnected from the server.
FTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in *lParam* will reference the defunct server and cannot be used with other server methods. The *lParam* message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The FTP_NOTIFY_CONNECT and FTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **GetClientServer** method to obtain a handle to the server that created the client session. Note that at the time the application processes the FTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

GetClientServer, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::AttachHandle Method

```
VOID AttachHandle(
   HSERVER hServer
);
```

The **AttachHandle** method attaches the specified server handle to the current instance of the class.

Parameters

hServer

The handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle has already been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the ***hServer*** parameter is presumed to be a valid server handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid server handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::AuthenticateClient Method

```
BOOL AuthenticateClient(
   UINT nClientId,
   DWORD dwUserAccess,
   BOOL bCreateHome,
   LPCTSTR lpszDirectory
);
```

Authenticate the client and assign access rights for the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User Access Constants.

bCreateHome

An integer value that specifics if the server should create the home directory for the authenticated client if it does not already exist. If this value is non-zero, the home directory will be created. If value is zero, the home directory will not be created and if it does not exist, this function will fail.

lpszDirectory

A null-terminated string that specifies the home directory for the user. If an absolute path is specified, it will be relative to the server root directory. If a relative path is specified, it will be a subdirectory of the home directory for the server instance. If this parameter is NULL or an empty string, a home directory will be assigned based on the server home directory and the user name.

Return Value

If the client session could be authenticated, the return value is non-zero. If the client ID does not specify a valid client session, or the client has already been authenticated, this method will return zero.

Remarks

The **AuthenticateClient** method is used to authenticate a specific client session, typically in response to an **OnAuthenticate** event that indicates a client has requested authentication. This method is also used internally to automatically grant the appropriate access rights to local user and anonymous client sessions.

It is recommended that most applications specify FTP_ACCESS_DEFAULT as the ***dwUserAccess*** value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options for normal and anonymous users. If the server is going to be publicly accessible or third-party FTP clients will be used to access the server, you should always grant the FTP_ACCESS_LIST permission to clients. Many client applications will not function correctly if they are unable to obtain a list of files in the user's home directory.

If FTP_ACCESS_RESTRICTED is specified and the server was started in multi-user mode, the client session will be effectively locked to its home directory and cannot navigate to the server root directory. By default, restricted client sessions are also limited to only downloading files and requesting directory listings. If a client session is not restricted, the client can access files outside of its home directory. Regardless of this option, a client cannot access files outside of the server root directory.

If FTP_ACCESS_RESTRICTED or FTP_ACCESS_ANONYMOUS is specified, the client session will be authenticated in a restricted mode and the access rights for the session will persist until the client disconnects from the server. Unlike regular users, the access rights for a restricted client cannot be changed by the server at a later point. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

If the ***lpszDirectory*** parameter is NULL or an empty string and the server has been started in multi-user mode, each user is assigned their own home directory based on their username. If the server has not been started in multi-user mode, then the default home directory will be the server root directory and is shared by all users. The **GetClientHomeDirectory** method will return the full path to the home directory for an authenticated client.

If the FTP_ACCESS_EXECUTE permission is granted to the client session, it can execute external programs using the SITE EXEC command. Because the program is executed in the context of the server process, it is recommended that you limit access to this functionality and ensure that the programs being executed do not introduce any security risks to the operating system. This permission is never granted by default, and the SITE EXEC command will return an error if the client session is anonymous, regardless of whether this permission is granted or not.

This method is should only be used for custom authentication schemes and is not necessary if you have used the **AddVirtualUser** method to create virtual users.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualUser, ChangeClientDirectory, GetClientCredentials, GetClientDirectory, OnAuthenticate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::CFtpServer Method

```
CFtpServer();
```

The **CFtpServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **FtpServerInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** method because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CFtpServer, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::ChangeClientDirectory Method

```
BOOL ChangeClientDirectory(
   UINT nClientId,
   LPCTSTR lpszDirectory
);
```

Change the current working directory for the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string which specifies the new current working directory for the client session. If this parameter is NULL or an empty string, the current working directory will be changed to the client home directory. If this parameter is not NULL, it must specify a directory that exists and is accessible by the server process.

Return Value

If the current working directory was changed, the return value is non-zero. If the client ID does not specify a valid client session, or the directory is invalid, this method will return zero.

Remarks

The **ChangeClientDirectory** method will change the current working directory for the specified client session. This method is called internally when the client sends the CWD or CDUP commands, however it may be explicitly used by the application to change the client's working directory in response to a server event.

This method cannot be used to change the current working directory for a client to an arbitrary directory outside of the server root directory. If the *lpszDirectory* parameter specifies a relative path (i.e.: a path that does not begin with a drive letter or leading path delimiter) then the new working directory will be relative to the current working directory. If an absolute path is specified, the absolute path must include the complete path to either the server root directory or the user's home directory, based on the permissions granted to the client session. If a path outside of the server root directory is specified, this method will fail with an access denied error.

Use caution when calling this method to override the directory specified by the client when it sends the CWD or CDUP commands. If your application changes the current working directory to one not specified by the client, it may cause unpredictable behavior in the client application because the actual path of the current working directory will not match the directory that was requested.

If this method is used to change the current working directory in response to the CWD command, you should not call the **GetCommandParam** method and pass the command parameter as an argument to this method. You must use the **GetCommandFile** method to obtain the directory name provided by the client prior to calling this method.

The application should never call the **SetCurrentDirectory** function in the Windows API to change the current directory for the process to the working directory of a client session. Because the server is multithreaded and each client session is managed in its own thread, an application using this library should avoid using relative paths.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientDirectory, GetClientHomeDirectory, GetCommandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::DeleteVirtualUser Method

```
BOOL DeleteVirtualUser(
   UINT nHostId,
   LPCTSTR lpszUserName
);
```

Remove a virtual user from the specified host.

Parameters

nHostId

An integer value which identifies the virtual host. This parameter is reserved for future use and must always have a value of zero.

lpszUserName

A null-terminated string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This method removes a virtual user that was created by a previous call to the **AddVirtualUser** method. This method will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive. You cannot use this method to delete the "anonymous" user.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::DetachHandle Method

```
HSERVER DetachHandle();
```

The **DetachHandle** method detaches the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If there is no active server, the value INVALID_SERVER will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketTools API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::DisableCommand Method

```
BOOL DisableCommand(
   LPCTSTR lpszCommand
);
```

Disable a specific server command.

Parameters

lpszCommand

A null-terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the command is not recognized, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **DisableCommand** method is used to disable access to a specific command on the server. When a command is disabled, it will also disable any corresponding feature related to that command. For example, if the MDTM command is disabled and a client issues the FEAT command to request a list of supported features, the command will no longer be listed. This method is typically used to disable certain commands for compatibility with older client software. The **IsCommandEnabled** method can be used to determine if a command is enabled or not.

The command name provided to this method must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would bet RETR (retrieve) and STOR (store). Refer to File Transfer Protocol Commands for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the USER and PASS commands are required to perform client authentication and therefore cannot be disabled. If you attempt to disable a required command, this method will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this method affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, either assign the client the appropriate permissions using the **AuthenticateClient** method, or use an event handler to filter the commands.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, EnableCommand, IsCommandEnabled

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::DisableTrace Method

```
BOOL DisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::DisconnectClient Method

```
BOOL DisconnectClient(
   UINT nClientId
);
```

Close the control connection for the specified client and release the resources allocated for the session

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **DisconnectClient** method will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted. This performs the same operation as if the client sent the QUIT command to the server.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active file transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

After this method returns, the application should never use the same client ID with another method. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

Restart, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::EnableClientAccess Method

```
BOOL EnableClientAccess(
   UINT nClientId,
   DWORD dwUserAccess,
   BOOL bEnable
);
```

Enable or disable access rights for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see User Access Constants.

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the ***dwUserAccess*** parameter. If this value is zero, that permission is revoked.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **EnableClientAccess** method is used to enable or disable access to specific functionality by the client. The method can only change a single access right and cannot be used to enable or disable multiple access rights in a single method call. To change multiple user access rights for the client, use the **SetClientAccess** method.

This method cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

AuthenticateClient, GetClientAccess, SetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::EnableCommand Method

```
BOOL EnableCommand(
   LPCTSTR lpszCommand
);
```

Enable a specific server command.

Parameters

lpszCommand

A null-terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the command is not recognized, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **EnableCommand** method is used to enable access to a specific command on the server. When a command is enabled, it will also enable any corresponding feature related to that command. For example, if the MDTM command is enabled and a client issues the FEAT command to request a list of supported features, the command will be included in the list. The **IsCommandEnabled** method can be used to determine if a command is enabled or not.

The command name provided to this method must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would bet RETR (retrieve) and STOR (store). Refer to File Transfer Protocol Commands for a complete list of server commands.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, DisableCommand, IsCommandEnabled

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the process level and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "`0x`". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::EnumClients Method

```
INT EnumClients(
   UINT *lpClients,
   INT nMaxClients
);
```

Return a list of active client connections established with the server.

Parameters

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the ***lpClients*** array. If the ***lpClients*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections, the method will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the method with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero.

Example

```
// Populate a listbox with all of the users connected to the server
pListBox->ResetContent();

INT nClients = pFtpServer->EnumClients();
if (nClients > 0)
{
    UINT *pIdList = new UINT[nClients];

    nClients = pFtpServer->EnumClients(pIdList, nClients);
    if (nClients == FTP_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        CString strUserName;

        if (pFtpServer->GetClientUserName(pIdList[nIndex], strUserName))
            pListBox->AddString(strUserName);
    }

    // Free the memory allocated for the client IDs
```

```
    delete pIdList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetClientAddress, GetClientDirectory, GetClientUserName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetActiveClient Method

```
UINT GetActiveClient();
```

Return the client ID for the active client session associated with the current thread.

Parameters

None.

Return Value

If the method succeeds, the return value is the unique ID associated with the client session for the current thread. If there is no client session active on the current thread, the return value is zero.

Remarks

The **GetActiveClient** method is used to obtain the client ID associated with the current thread. This means this method will only return a client ID if it is called within an event handler or a method called by an event handler. If this method is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

EnumClients

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetAddress Method

```
INT GetAddress(
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetAddress(
   UINT nClientId,
   CString& strAddress
);
```

Return the IP address of the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A null-terminated string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

This method will return the IP address assigned to the specified server as a printable string. If the ***nClientId*** parameter has a value of zero, this method will return the IP address assigned to the local system. If the FTP_SERVER_NATROUTER option was specified when the server was started, this method will return the external IP address assigned to the system. If the ***nClientId*** parameter specifies a valid client session, this method will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **GetClientAddress** method.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetName, SetAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientAccess Method

```
BOOL GetClientAccess(
   UINT nClientId,
   DWORD& dwUserAccess
);
```

Return the access rights that have been granted to the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see User Access Constants.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **GetClientAccess** method is used to obtain all of the access rights that are currently granted to an authenticated client session. The **EnableClientAccess** method can be used to enable or disable specific permissions, and the **SetClientAccess** method can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;

// Check if the client is a restricted user
if (pFtpServer->GetClientAccess(nClientId, dwUserAccess))
{
    if (dwUserAccess & FTP_ACCESS_RESTRICTED)
    {
        std::cout << "Client authenticated as a restricted user\n";
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

AuthenticateClient, EnableClientAccess, SetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientAddress Method

```
INT GetClientAddress(
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetClientAddress(
   UINT nClientId,
   CString& strAddress
);
```

Return the IP address of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A null-terminated string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientDirectory, GetClientUserName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientCredentials Method

```
BOOL GetClientCredentials(
   UINT nClientId,
   LPFTPCLIENTCREDENTIALS lpCredentials
);

BOOL GetClientCredentials(
   UINT nClientId,
   CString& strUserName,
   CString& strPassword
);
```

Return the user credentials for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **FTPCLIENTCREDENTIALS** structure that will contain information about the user when the method returns. This parameter cannot be NULL.

strUserName

A string that will contain the user name when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

strPassword

A string that will contain the user password when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not requested authentication, this method will return zero.

Remarks

The **GetClientCredentials** method is used to obtain the username and password that was provided by the client when it requested authentication. Typically this method is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **AuthenticateClient** method to specify that the session has been authenticated.

If the default event handler is used, the **OnAuthenticate** method will be invoked with the user credentials passed to the handler as arguments.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, OnAuthenticate, FTPCLIENTCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientDirectory Method

```
INT GetClientDirectory(
   UINT nClientId,
   LPTSTR lpszDirectory,
   INT nMaxLength
);

INT GetClientDirectory(
   UINT nClientId,
   CString& strDirectory
);
```

Returns the current working directory for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string buffer that will contain the current working directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero.

Remarks

This method returns the full path to the current working directory for the specified client session. For example, if the server root directory is C:\ProgramData\MyServer and the current working directory for the client is /Research/Documents, this method will return C:\ProgramData\MyServer\Research\Documents as the current working directory for the client session.

It is important to note that the current working directory for client sessions is virtual, and does not reflect the current working directory for the server process. To change the current working directory for a client, use the **ChangeClientDirectory** method.

This method should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a current working directory and this method will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

To convert a full path to the virtual path for a specific client session, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ChangeClientDirectory, GetClientHomeDirectory, GetClientVirtualPath,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientFileType Method

```
BOOL GetClientFileType(
   UINT nClientId,
   UINT& nFileType
);
```

Return the current file type used for transfers by the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpnFileType

An unsigned integer value that will contain the current file type used by the client for data transfers.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetClientFileType** method will return the current file type that has been specified by the client sending the TYPE command to the server. The file type determines if there is any conversion performed on the data that is being exchanged between the client and server. The following file types are supported:

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

SetClientFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientHomeDirectory Method

```
INT GetClientHomeDirectory(
   UINT nClientId,
   LPTSTR lpszDirectory,
   INT nMaxLength
);

INT GetClientHomeDirectory(
   UINT nClientId,
   CString& strDirectory
);
```

Returns the home directory for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string buffer that will contain the home directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero.

Remarks

This method returns the full path to the home working directory assigned to the specified client session. This will be the same path to the home directory specified when the **AuthenticateClient** method was used to authenticate the client session. If a home directory was not explicitly assigned when the client was authenticated, then this method returns the default home directory that was created for the client, or the server root directory if the FTP_SERVER_MULTIUSER option was not specified when the server was started.

This method should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a home directory and this method will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, ChangeClientDirectory, GetClientDirectory, GetClientVirtualPath

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientIdentity Method

```
INT GetClientIdentity(
   UINT nClientId,
   LPTSTR lpszIdentity,
   INT nMaxLength
);

INT GetClientIdentity(
   UINT nClientId,
   CString& strIdentity
);
```

Return the identity of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A null-terminated string buffer that will contain the identity of the client when the method returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid, or the buffer is not large enough to store the complete path, the method will return a value of zero. If the client did not identify itself, this method will return zero.

Remarks

The **GetClientIdentity** method returns the string that the client used to identify itself to the server. The client may use either the CLNT or CSID command to identify itself. Although the CLNT command is considered to be deprecated, it is supported for backwards compatibility with older clients. The identity string does not have any standard format and is used for informational purposes only and does not affect the operation of the server in any way. Not all clients identify themselves, in which case this method will return zero and the ***lpszIdentity*** string buffer will be set to an empty string.

If the client does identify itself, it typically uses the name of the client application that was used to establish the connection. The application may choose to assign an identity to a client session for its own internal purposes using the **SetClientIdentity** method, regardless of whether the client identifies itself.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetIdentity, SetClientIdentity, SetIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientIdleTime Method

```
UINT GetClientIdleTime(
   UINT nClientId,
   UINT *lpnElapsed
);
```

Return the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

An optional pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL or omitted if this information is not required.

Return Value

If the method succeeds, the return value is client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetClientIdleTime** method will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a file data transfer completed. The server will never disconnect a client that is in the process of uploading or downloading a file, regardless of the idle timeout period.

The default idle timeout period for a client is 900 seconds (15 minutes), however the server can be configured to use a different value and individual clients can request the timeout period be changed by sending the IDLE command to the server. For a client to be able to change its own timeout period, it must be granted the FTP_ACCESS_IDLE permission. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). An application can change the timeout period for a specific client session using the **SetClientIdleTime** method.

It is important to note that the idle timeout period only affects authenticated clients. Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server before successfully authenticating with a valid username and password. By default, the authentication timeout period is 60 seconds and is set when the server is started; it cannot be changed for an individual client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

SetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientLocalPath Method

```
INT GetClientLocalPath(
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   LPTSTR lpszLocalPath,
   INT nMaxLength,
);

INT GetClientLocalPath(
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   CString& strLocalPath
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A null-terminated string that specifies an virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A null-terminated string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **GetClientLocalPath** method takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the current working directory for the client session. This method will recognize a tilde at the beginning of the path to specify the client home directory.

To obtain the virtual path for a local file or directory, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientServer Method

```
HSERVER HttpGetClientServer(
    UINT nClientId
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SERVER. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return INVALID_SERVER indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

AsyncNotify

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientThreadId Method

```
DWORD GetClientThreadId(
   UINT nClientId
);
```

Returns the thread ID associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **DisconnectClient** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

EnumClients, GetActiveClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientUserName Method

```
INT GetClientUserName(
   UINT nClientId,
   LPTSTR lpszUserName,
   INT nMaxLength
);

INT GetClientUserName(
   UINT nClientId,
   CString& strUserName
);
```

Return the user name associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

strUserName

A null-terminated string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not authenticated itself, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, GetClientAccess, GetClientHomeDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientVirtualPath Method

```
INT GetClientVirtualPath(
   UINT nClientId,
   LPCTSTR lpszLocalPath,
   LPTSTR lpszVirtualPath,
   INT nMaxLength,
);

INT GetClientVirtualPath(
   UINT nClientId,
   LPCTSTR lpszLocalPath,
   CString& strVirtualPath
);
```

Return the virtual path for a local file on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A null-terminated string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A null-terminated string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is either relative to the server root directory, or the client home directory if the client was authenticated as a restricted user. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the ***lpszLocalPath*** parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this method would return the virtual path to that directory as "/Documents/Research".

If the client session was authenticated as a restricted user, then the virtual path is always relative to the client home directory instead of the server root directory. This is because restricted users are isolated to their own home directory and any subdirectories. For example, if restricted user "John" has a home directory of "C:\ProgramData\MyServer\Users\John" and the ***lpszLocalPath*** parameter was specified

as "C:\ProgramData\MyServer\Users\John\Accounting\Projections.pdf" this method would return the virtual path as "/Accounting/Projections.pdf".

If the ***lpszLocalPath*** parameter specifies a file or directory outside of the server root directory, this method will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This method can only be used with authenticated clients. If the ***nClientId*** parameter specifies a client session that has not been authenticated, this method will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetClientLocalPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandFile Method

```
INT GetCommandFile(
   UINT nClientId,
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetCommandFile(
   UINT nClientId,
   CString& strFileName
);
```

Get the full path to the local file name or directory specified by the client

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null-terminated and must be large enough to store the complete file path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that the command issued by the client accepts a file name as an argument, but the client did not specify one.

Remarks

The **GetCommandFile** method is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard FTP command. For example, if the client sends the RETR command to the server, this method will return the complete path to the local file that the client wants to download. This method will only work with those standard commands that perform some action on a file or directory.

This method should always be used to obtain the file name for a command that performs a file or directory operation. The **GetCommandParam** method will return the actual command parameter, but the file name will typically be relative to the user home directory or server root directory, and cannot be passed directly to a Windows API function. The **GetCommandFile** method normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location.

To change the file or directory name that is the target of the current command, use the **SetCommandFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientDirectory, GetClientHomeDirectory, GetCommandLine, GetCommandParam, SetCommandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandLine Method

```
INT GetCommandLine(
   UINT nClientId,
   LPTSTR lpszCmdLine,
   INT nMaxLength
);

INT GetCommandLine(
   UINT nClientId,
   CString& strCmdLine
);
```

Return the complete command line issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A null-terminated string buffer that will contain the command, including all arguments. The string buffer will be null-terminated and must be large enough to store the complete command line. If this parameter is NULL, the method will return the length of the command line. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **GetCommandLine** method is used to obtain the command that was issued by the client, and is commonly used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. When the method returns, the string buffer provided by the caller will contain the complete command, including all command parameters. Any extraneous whitespace will be removed, however quoted parameters will be retained as-is.

To obtain a specific parameter to a command, use the **GetCommandParam** method. The **GetCommandParamCount** method will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **GetCommandFile** method should be called to obtain the full path to the specified file rather than using the value of the command parameter.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandParam, OnCommand, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandName Method

```
INT GetCommandName
   UINT nClientId,
   LPTSTR lpszCommand,
   INT nMaxLength
);

INT GetCommandName
   UINT nClientId,
   CString& strCommand
);
```

Return the name of the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string buffer that will contain the command name. The string buffer will be null-terminated and must be large enough to store the complete command name. If this parameter is NULL, the method will only return the length of the current command in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszCommand* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **GetCommandName** method is used to obtain the name of the last command that was issued by the client. The command name returned by this method will always be capitalized, regardless of how it was sent by the client. This method is typically used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively.

The **GetCommandParam** method can be used to return the value of individual command parameters specified by the client. The **GetComandLine** method can be used to obtain the complete command line issued by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandLine, GetCommandParam, OnCommand, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandParam Method

```
INT GetCommandParam
   UINT nClientId,
   INT nParam,
   LPTSTR lpszParam,
   INT nMaxLength
);

INT GetCommandParam
   UINT nClientId,
   INT nParam,
   CString& strParam
);
```

Return the value of the specified command parameter from the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nParam

An integer value which specifies the command parameter. A value of zero specifies the command itself, while values greater than zero specify a particular parameter. This method will fail if this value is less than zero or greater than the number of parameters available.

lpszParam

A null-terminated string buffer that will contain the command parameter. The string buffer will be null-terminated and must be large enough to store the complete parameter value. If this parameter is NULL, the method will only return the length of the specified parameter in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszParam* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **GetCommandParam** method is used to obtain a specific parameter for the last command that was issued by the client. If the parameter was surrounded in quotes, those quotes will be included in the value returned by this method. This method is typically used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively.

The **GetCommandParamCount** method will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or

directory, the **GetCommandFile** method should be called to obtain the full path to the specified file rather than using the value of the command parameter. To obtain the complete command line, use the **GetCommandLine** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandLine, GetCommandName, GetCommandParamCount, OnCommand, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandParamCount Method

```
INT GetCommandParamCount
   UINT nClientId
);
```

Return the number of command parameters for the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An integer value which specifies the number of parameters that were specified in the last command issued by the client. If the command did not include any parameters, this method will return zero. If the client has not issued a command, or the client session ID is invalid, this method will return -1.

Remarks

The **GetCommandParamCount** method is used to determine the number of parameters specified in the last client command, and the maximum value that may be passed as the parameter index to the **GetCommandParam** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetCommandLine, GetCommandName, GetCommandParam

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandResult Method

```
INT GetCommandResult(
   UINT nClientId,
   LPTSTR lpszResult,
   INT nMaxLength
);

INT GetCommandResult(
   UINT nClientId,
   CString& strResult
);
```

Return the result code and description for the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A null-terminated string buffer that will contain the description of the result code. The string buffer will be null-terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **GetCommandResult** method is used to determine the result of the last command that was issued by the client and is typically called in the **OnResult** event handler. This method should only be called after a command has been processed or the **SendResponse** method has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code, and optionally a text message that further describes the result. The text message may be a single line, or it may span multiple lines, with each line of text terminated by a carriage return and linefeed. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range indicate that the requested action is being initiated, and the client should expect another reply from the server before proceeding. This is normally used with file transfers, indicating to the client that the data transfer has started.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. One exception is the 202 result code which indicates that the command is not implemented, but the client should not consider this to be an error condition.

300-399	Result codes in this range indicate that the requested action cannot complete until additional information is provided to the server. This is normally used with commands that require a specific sequence to complete. For example, the server will send the 331 result code in response to the USER command, which tells the client that it must send the PASS command to complete the authentication process.
400-499	Result codes in this range indicate that the requested action did not take place, but the error condition is temporary and may be attempted again. This error response is usually the result of a failed authentication attempt or a file transfer that could not complete.
500-599	Result codes in this range indicate that the requested action did not take place and the failure is permanent. The client should not attempt to send the command again. This error response is usually the result of an invalid command name, a syntax error or the client not having the appropriate access rights to a resource.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SendResponse, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetCommandUsage Method

```
UINT GetCommandUsage(
   LPCTSTR lpszCommand
);
```

Return the number of times a specific command has been issued by all clients.

Parameters

lpszCommand

A null-terminated string that specifies a command name. The name is not case-sensitive, but must match a valid server command exactly. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is the number of times the command has been issued by all clients since the server was started. If the command name is invalid, or the command has never been issued, the return value will be zero.

Remarks

The command name provided to this method must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would bet RETR (retrieve) and STOR (store). Refer to File Transfer Protocol Commands for a complete list of server commands.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

EnableCommand, GetCommandLine, GetCommandResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetDirectory Method

```
INT GetDirectory(
   LPTSTR lpszDirectory,
   INT nMaxLength
);

INT GetDirectory(
   CString& strDirectory
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

lpszDirectory

A null-terminated string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetDirectory** method will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, the current working directory will be used and this method can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This method will return the fully resolved path name, with all environment variables expanded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetIdentity, GetName, SetDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetHandle Method

```
HSERVER GetHandle();
```

The **GetHandle** method returns the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If the server is inactive, the value INVALID_SERVER will be returned.

Remarks

This method is used to obtain the server handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetIdentity Method

```
INT GetIdentity(
   LPTSTR lpszIdentity,
   INT nMaxLength
);

INT GetIdentity(
   CString& strIdentity
);
```

Return the identity of the specified server.

Parameters

lpszIdentity

A null-terminated string buffer that will contain the identity of the server when the method returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetIdentity** method returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number, and is displayed whenever a client establishes its initial connection to the server. The **SetIdentity** method can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientIdentity, SetClientIdentity, SetIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetLastError Method

```
DWORD GetLastError(
   LPTSTR lpszError,
   INT nMaxLength
);

DWORD GetLastError(
   CString& strError
);

DWORD GetLastError();
```

Return the last server error code and a description of the error.

Parameters

lpszError

A null-terminated string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the ***lpszError*** parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods clear the last error code when they succeed.

It is important to note that the error codes returned by this method are different than the command result codes that are defined in RFC 959, the standard protocol specification for FTP. This method is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **SendResponse** method is used to send result codes to the client, and the **GetCommandResult** method can be used to determine the result of the last command sent by the client.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR. Those methods which clear the last error code when they succeed are noted on their reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandResult, SendResponse, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetLogFile Method

```
BOOL GetLogFile(
   UINT *lpnLogFormat,
   UINT *lpnLogLevel,
   LPTSTR lpszFileName,
   INT nMaxLength
);

BOOL GetLogFile(
   UINT *lpnLogFormat,
   UINT *lpnLogLevel,
   CString& strFileName
);

BOOL GetLogFile(
   CString& strFileName
);
```

Return the current log file format and the full path to the file.

Parameters

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the method returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Value	Description
FTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A null-terminated string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the *lpszFileName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **FTPSERVERCONFIG** structure definition for a list of supported log file formats. If logging has not been enabled, this method will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This method can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **RenameServerLogFile** method to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **SetLogFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RenameServerLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero and specifies the amount of memory allocated by the server. If the server is inactive or cannot be locked, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the server into a locked state, and all client sessions will block until the method returns. Because this method enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this method can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetStackSize, SetStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetName Method

```
INT GetName(
   UINT nClientId,
   LPTSTR lpszHostName,
   INT nMaxLength
);

INT GetName(
   UINT nClientId,
   CString& strHostName
);
```

Return the host name assigned to the specified server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will return the host name assigned to the specified server. If the ***nClientId*** parameter has a value of zero, the method will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the ***nClientId*** parameter specifies a client session, then it this method will return the host name that the client used to establish the connection. If the client sends the HOST command to the server, this method will return the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetOptions Method

```
DWORD GetOptions();
```

Return the options specified for this instance of the server.

Parameters

None.

Return Value

The current server options. For a list of available options, see Server Option Constants

Remarks

The **GetOptions** method returns the default options for the current instance of the server. To change the server options, use the **SetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetPriority Method

```
INT GetPriority();
```

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Value	Description
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

SetPriority, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetProgramExitCode Method

```
BOOL GetProgramExitCode(
   UINT nClientId,
   DWORD& dwExitCode
);
```

Return the exit code of the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwExitCode

An unsigned integer that will contain the program exit code when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetProgramExitCode** method returns the exit code of a registered program that was executed by the client using the SITE EXEC command. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by custom programs to communicate status information back to the server application.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetProgramName, GetProgramOutput, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetProgramName Method

```
INT GetProgramName(
   UINT nClientId,
   LPTSTR lpszProgramName,
   INT nMaxLength
);

INT GetProgramName(
   UINT nClientId,
   CString& strProgramName
);
```

Return the name of the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A null-terminated string buffer that will contain the name of the last program executed by the client. This parameter cannot be NULL and should be at least 32 characters in size. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the method succeeds, the return value is the length of the command name. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, this method will return zero.

Remarks

The **GetProgramName** method returns the name of the last program that was executed by the client using the SITE EXEC command. The name that is returned is the alias assigned to the program, not the full path to the executable file. The server application would typically use this method in an event handler when processing the FTP_CLIENT_EXECUTE event to determine which program has been executed on behalf of the client. The **GetProgramExitCode** method will return the program's exit code and the **GetProgramOutput** method can be used to obtain a copy of the output generated by the program.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramOutput, OnExecute, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetProgramOutput Method

```
DWORD GetProgramOutput(
   UINT nClientId,
   LPBYTE lpBuffer,
   DWORD dwBufferSize
);
```

Return a copy of the standard output from the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the method will return the number of bytes of data that was output by the program. Note that this output is not null-terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the ***lpBuffer*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of bytes copied into the specified buffer. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramOutput** method is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of bytes of output available to read, call this method with the ***lpBuffer*** parameter as NULL and the ***dwBufferSize*** parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null-terminated.

This method returns the raw output from the command which may contain escape sequences, control characters and embedded nulls. When the application processes the output returned by this method, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the command as a string, use the **GetProgramText** method.

Example

```
LPBYTE lpBuffer = NULL;  // The output buffer
DWORD cbBuffer = 0;      // Number of bytes in the output buffer

// Determine the number of bytes in the output buffer
cbBuffer = pFtpServer->GetProgramOutput(nClientId, NULL, 0);

if (cbBuffer > 0)
{
    // Allocate memory for the buffer
    lpBuffer = new BYTE[cbBuffer + 1];
```

```
    // Copy the program output to the buffer
    cbBuffer = pFtpServer->GetProgramOutput(nClientId, lpBuffer, cbBuffer + 1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    delete lpBuffer;
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetProgramExitCode, GetProgramName, GetProgramText, OnExecute, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetProgramText Method

```
INT GetProgramText(
   UINT nClientId,
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT GetProgramText(
   UINT nClientId,
   CString& strBuffer
);
```

Return a copy of the standard output from the last program in a string buffer.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the method will return the number of bytes of characters that was output by the program, not including a terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the ***lpszBuffer*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramText** method is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of characters of output available to read, call this method with the ***lpszBuffer*** parameter as NULL and the ***nMaxLength*** parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This method will only return textual output from the command and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the last command that was executed, use the **GetProgramOutput** method.

Example

```
CString strBuffer;

if (pFtpServer->GetProgramText(nClientId, strBuffer) > 0)
    pEditCtrl->SetWindowText(strBuffer);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramName, GetProgramOutput, OnExecute, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetRenamedFile Method

```
INT GetRenamedFile(
   UINT nClientId,
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetRenamedFile(
   UINT nClientId,
   CString& strFileName
);
```

Return the original name of a file being renamed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string buffer that will contain the full path of the last file or directory that was renamed. It is recommended that this buffer be at least MAX_PATH characters in size. The value of this parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null characters. This parameter must have a value larger than zero.

Return Value

If the method succeeds, the return value is the length of the file or directory path, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the client has not renamed a file or directory, this method will return zero.

Remarks

When a client wishes to rename a file or directory, it must send two commands in sequence to the server. The first command is RNFR (rename from) which specifies the original name of the file or directory to be renamed. The second command is RNTO (rename to) and must be sent immediately after the RNFR command and specifies the new name for the file or directory. The **GetRenamedFile** method will return the full path of the local file or directory that was specified by the RNFR command. Typically this method is used by an event handler that processes the **OnCommand** event to determine the original path name.

This method is only guaranteed to return a meaningful value when called within the context of the **OnCommand** event handler. Calling this method outside of the event handler will return the path of the last renamed file, but there is no way to determine at what point the client issued the command to rename a file or directory. To obtain the new file or directory name, the **GetCommandFile** method should be called from within the event handler.

The RNFR and RNTO commands can also be used to move a file or directory to a new location. For example, they could be used to move a file from one directory to another. An application should never make the assumption that the paths of the original and new file will be the same.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, OnCommand

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetStackSize Method

```
DWORD GetStackSize();
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

SetStackSize, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetTransferInfo Method

```
BOOL GetTransferInfo(
   UINT nClientId,
   LPFTPSERVERTRANSFER lpTransferInfo
);
```

Return information about the current file transfer for the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **FTPSERVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the ***dwSize*** member of the structure must be initialized to specify the structure size prior to calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method should only be called after the client has issued the APPE, RETR, STOR or STOU commands to initiate a file transfer, otherwise the return value will be zero.

Remarks

The **GetTransferInfo** method is used to obtain information about the last file transfer that was performed by the client. This method is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

If the default event handler is used, the **OnDownload** and **OnUpload** methods will be invoked with information about the transfer passed as arguments to the handler.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandResult, OnDownload, OnUpload, FTPSERVERTRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetUuid Method

```
INT GetUuid(
   LPTSTR lpszHostUuid,
   INT nMaxLength
);

INT GetUuidString(
   CString& strHostUuid
);
```

Return the UUID assigned to the server as a printable string.

Parameters

lpszHostUuid

A null-terminated string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete UUID string, the method will return a value of zero.

Remarks

The **GetUuid** method returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetUuid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::IsActive Method

```
BOOL IsActive();
```

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

CFtpServer, IsListening, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::IsClientAnonymous Method

```
BOOL IsClientAnonymous(
   UINT nClientId
);
```

Determine if the specified client has authenticated as an anonymous user.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this method will return a non-zero value, otherwise it will return zero. If the client ID is valid, and the client session has been authenticated, this method will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

AuthenticateClient, GetClientCredentials, IsClientAuthenticated

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::IsClientAuthenticated Method

```
BOOL IsClientAuthenticated(
    UINT nClientId
);
```

Determine if the specified client session has been authenticated.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this method will return a non-zero value, otherwise it will return zero. If the client ID is valid, this method will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

AuthenticateClient, GetClientCredentials, IsClientAnonymous

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::IsCommandEnabled Method

```
BOOL IsCommandEnabled(
   LPCTSTR lpszCommand
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

lpszCommand

A null-terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the command is enabled, this method will return a non-zero value. If the command is disabled or the command name does not match a supported command, this method will return zero.

Remarks

The **IsCommandEnabled** method is used to determine whether a specific command is enabled. Typically this method is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled using the **EnableCommand** method and disabled using the **DisableCommand** method.

This method does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **AuthenticateClient** method, and certain commands can be limited by the permissions granted to the client. For example, even though the STOR command is enabled, a client must have the FTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see User Access Constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, DisableCommand, EnableCommand, GetCommandName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::IsListening Method

```
BOOL IsListening();
```

The **IsListening** method returns whether or not the server is listening for client connections.

Return Value

This method returns a non-zero value if the server has been started and is listening for client connections. If the server is stopped or has been suspended this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

CFtpServer, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(
   HSERVER hServer,
   UINT nClientId,
   UINT nEventId,
   DWORD dwError,
   BOOL& bHandled
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see Server Event Constants.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::RegisterProgram Method

```
BOOL RegisterProgram(
   LPCTSTR lpszCommandName,
   LPCTSTR lpszProgramFile
);

BOOL RegisterProgram(
   LPCTSTR lpszCommandName,
   LPCTSTR lpszProgramFile,
   LPCTSTR lpszParameters,
   LPCTSTR lpszDirectory,
);
```

Register a program for use with the SITE EXEC command.

Parameters

lpszCommandName

A null-terminated string which specifies the name of the site specific command. This is the name that is passed to the SITE EXEC command and does not need to match the actual name of the executable file on the local system. The maximum length of the command name is 32 characters. This parameter cannot be NULL.

lpszProgramFile

A null-terminated string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A null-terminated string that specifies additional parameters for the program. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A null-terminated string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the current working directory of the client that issues the SITE EXEC command.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **RegisterProgram** method registers an executable program for use with the SITE EXEC command. Because this can present a significant security risk to the server, clients are not given permission to use this command by default. A client must be explicitly granted permission to use SITE EXEC by including FTP_ACCESS_EXECUTE as one of the permissions when authenticating the client session with the **AuthenticateClient** method or creating a virtual user using the **AddVirtualUser** method.

To give the server complete control over what programs can be executed using SITE EXEC, the program must be registered with the server and referenced by an alias specified by the *lpszCommandName* parameter. The maximum length of a program name is 31 characters and it must be at least 3 characters in length. The name must only consist of alphanumeric characters and the first character of the program name cannot be numeric. The program name is not case-sensitive, however convention is to use upper-case characters. If a program name is specified that already has

been registered, it will be updated with the new information provided by this method.

The *lpszProgramFile* string specifies file name of the program that will be executed. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple command names that reference the same executable file. The only requirement is that the command names be unique. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string is used to define optional command line parameters that will be included with the command. This string can contain placeholders that are replaced by additional parameters specified by the client when it sends the SITE EXEC command. First replacement parameter is %1, the second is %2 and so on.

The executable program that is registered using this method must be a console application that writes to standard output. Programs that write directly to a console, or programs written to use a Windows user interface are not supported and will yield unpredictable results. In most cases, those programs that do not use standard input and output will be forcibly terminated by the server. If the program attempts to read from standard input, it will immediately encounter an end-of-file condition. Programs executed by the SITE EXEC command have no input; it is similar to a program that has its input redirected from the NUL: device. If the program must process a file on the server, the local file name should be passed as a command line parameter.

The output from the program will be redirected back to the client control channel. The output should be textual, with each line of text terminated by a carriage return and linefeed (CRLF). Programs that write binary data to standard output, particular data with embedded nulls, will yield unpredictable results and are not supported. To ensure that the program output conforms to the protocol standard, any non-printable characters will be replaced with a space and each line of output will be prefixed by a single space. The server application can obtain a copy of the output from the last command by calling the **GetProgramOutput** method.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramName, GetProgramOutput

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::RenameServerLogFile Method

```
BOOL RenameServerLogFile(
   LPCTSTR lpszFileName
);
```

Rename or delete the current log file being updated by the server.

Parameters

lpszFileName

A null-terminated string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If logging is not currently enabled for the server, this method will return zero.

Remarks

The **RenameServerLogFile** method is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the *lpszFileName* parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This method must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **GetLogFile** method to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** Windows API functions with that file.

To disable logging, use the **SetLogFile** method and specify the logging format as FTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::Restart Method

```
BOOL Restart();
```

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::Resume Method

```
BOOL Resume();
```

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

Restart, Start, Stop, Suspend, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SendResponse Method

```
BOOL SendResponse(
   UINT nClientId,
   UINT nResultCode,
   LPCTSTR lpszMessage
);
```

Send a result code and message to the client in response to a command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nResultCode

An unsigned integer value which specifies the result code.

lpszMessage

A null-terminated string which specifies a message to be sent to the client. If this parameter is NULL or points to an empty string, a default message associated with the result code will be used.

Return Value

If the result code and message text was sent to the client, the return value is non-zero. If the client ID does not specify a valid client session, or the result code is invalid, this method will return zero.

Remarks

The **SendResponse** method is used to respond to a command issued by the client. Command responses are normally handled by the server as a normal part of processing a command and this method is only used if the application has implemented custom commands or wishes to modify the standard responses sent by the server. The message may be a maximum of 2048 characters and may include embedded carriage-return and linefeed characters. If no message is specified, then a default message will be sent based on the result code.

Result codes must be three digits (in the range of 100 through 999) and although this method will support the use of non-standard result codes, it is recommended that the client application use the standard codes defined in RFC 959 whenever possible. The use of non-standard result codes may cause problems with FTP clients that expect specific result codes in response to a particular command. For more information, refer to the **GetCommandResult** method.

This method should only be called once in response to a command sent by the client. If a result code has already been sent in response to a command and this method is called, it will fail and return a value of zero. This is necessary because sending multiple result codes in response to a single command may cause unpredictable behavior by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetAddress Method

```
BOOL SetAddress(
   LPCTSTR lpszAddress
);
```

Change the IP address that the server will use with passive data connections.

Parameters

lpszAddress

A null-terminated string that specifies the IP address that the server should use for passive mode data transfers. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If either the server handle or the IP address is invalid the method will return a value of zero.

Remarks

The **SetAddress** method changes the IP address that the server will use when a client transfers a file using a passive mode data connection. In passive mode, the server will create a second passive (listening) socket that will accept an incoming connection from the client. The server sends the IP address and port number allocated for that socket to the client, and the client establishes the data connection to the server using that address. The IP address that the server sends to the client is normally the same as the IP address that the client used to establish the control connection, however if the server is located behind a router that performs Network Address Translation (NAT), the IP address reported to the client may not be usable.

This method enables your application to set the external IP address for the server to a specific value, rather than the server attempting to automatically discover its own external address. If you wish to set the external address for the server manually, call the **Start** method without the FTP_SERVER_EXTERNAL option and then call this method to set the external IP address to the desired value.

This method will not change the IP address the server is using to listen for client connections. The only way to change the listening IP address is to stop and restart the server using the new address. This method only changes the IP address that is reported to clients when a passive data connection is used. Incorrect use of this method can prevent the client from establishing a data connection to the server. The address must be in the same address family as the local address that the server was started with. For example, if the server was started using an IPv4 address, the IP address passed to this method cannot be an IPv6 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetCertificate Method

```
BOOL SetCertificate(
   DWORD dwProtocol,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);

BOOL SetCertificate(
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);
```

Set the name of the certificate to be used with secure connections.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpszPassword

An optional pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only required if the ***lpszCertStore*** parameter specifies a certificate file in PKCS #12 format that is password protected.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call the **GetLastError** method.

Remarks

An application only needs to call this method to support secure connections using TLS. When a certificate is specified, it must have a private key associated with it. Attempting to use a certificate which does not have a private key will result in an error during the startup process indicating that the server credentials are invalid.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the class will be unable to access the certificate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetOptions, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetClientAccess Method

```
BOOL SetClientAccess(
   UINT nClientId,
   DWORD dwUserAccess
);
```

Change the access rights associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User Access Constants.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **SetClientAccess** method can change the access rights for an authenticated client session. This method can only be used after the **AuthenticateClient** method has been used to grant the initial set of access rights to the client. The **EnableClientAccess** method can be used to grant or revoke a specific permission for the client session.

The *dwUserAccess* parameter has a value of FTP_ACCESS_DEFAULT, then default permissions will be granted to the client session. A normal client cannot be changed to a restricted or anonymous client using this method. If the FTP_ACCESS_RESTRICTED or FTP_ACCESS_ANONYMOUS access flags are specified, this method will fail.

This method cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Example

```
DWORD dwUserAccess = 0;

// Allow the client to execute programs using SITE EXEC
if (pFtpServer->GetClientAccess(nClientId, dwUserAccess))
{
    if (! (dwUserAccess & FTP_ACCESS_ANONYMOUS))
        pFtpServer->SetClientAccess(nClientId, dwUserAccess | FTP_ACCESS_EXECUTE);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

AuthenticateClient, EnableClientAccess, GetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetClientFileType Method

```
BOOL SetClientFileType(
   UINT nClientId,
   UINT nFileType
);
```

Change the current file type used for transfers by the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nFileType

Specifies the type of file that will be uploaded or downloaded. This parameter determines whether subsequent file transfers require any data conversion and may be one of the following values.

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetClientFileType** method will change the default file type that is used for subsequent transfers by the client. If the file type is set to FILE_TYPE_ASCII then the server will automatically convert any end-of-line character sequences to match the format used by the local system. For example, if the client is connecting from a UNIX based system, the server will convert a single linefeed character to a carriage return (CR) and linefeed (LF) sequence. If the file type is set to FILE_TYPE_IMAGE, then no conversion is performed.

This method can be used to override the file type specified by filtering the TYPE command issued by the client. For example, it could be used to force file transfers to use a specific type based on the file extension, regardless of the type specified by the client. To determine the current file type set by the client, use the **GetClientFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetClientFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetClientIdentity Method

```
BOOL SetClientIdentity(
   UINT nClientId,
   LPCTSTR lpszIdentity
);
```

Change the identity of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A null-terminated string that identifies the client. If this parameter is NULL or specifies an empty string, the current identity for the client is cleared. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the client ID is invalid, the method will return a value of zero.

Remarks

The **SetClientIdentity** method associates a string value with the client that can be used to identify the session. The identity string does not have any standard format and is used for informational purposes only. Typically it is used to identify the client application that was used to establish the connection. Changing the client identity has no effect on the operation of the server. To obtain the identity string currently associated with the client, use the **GetClientIdentity** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientIdentity, GetIdentity, SetIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetClientIdleTime Method

```
UINT SetClientIdleTime(
   UINT nClientId,
   UINT nTimeout
);
```

Change the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the method succeeds, the return value is the previous client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetClientIdleTime** method is will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an **OnTimeout** event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **GetClientIdleTime** method.

This timeout period only affects authenticated clients. Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server and that value cannot be changed for individual client sessions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetCommandFile Method

```
BOOL SetCommandFile(
   UINT nClientId,
   LPCTSTR lpszFileName
);
```

Change the name of the local file or directory that is the target of the current command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetCommandFile** method is used by the application to change the target file or directory name for the current command from within the **OnCommand** event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the RETR command to download a file from the server, this method can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **GetCommandFile** method.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is . If the path is relative, it will be relative to the current working directory for the client session. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this method to change the target file name.

Typically this method would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this method to provide access to data that is stored outside of the server root directory. Incorrect use of this method could expose the server to security risks or cause unpredictable behavior by client applications.

This method should only be called within the context of the **OnCommand** event handler, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this method will return zero and the last error code will be set to ST_ERROR_INVALID_COMMAND. To obtain the name of the current command issued by the client, use the **GetCommandName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile GetCommandLine, GetCommandName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetDirectory Method

```
BOOL SetDirectory(
   LPCTSTR lpszDirectory
);
```

Specify the local directory that will be used as the server root directory.

Parameters

lpszDirectory

A null-terminated string that specifies the root directory for the server. If this parameter is NULL or a zero-length string, the server will use the current working directory as the root directory.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero.

Remarks

The **SetDirectory** method specifies the path to the local directory that should be used as the root document directory for the server.

It is recommended that you always specify an absolute path for the server root directory. If the path includes environment variables surrounded by percent (%) symbols, they will be automatically expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

You cannot change the server root directory after the server has started. To change the root directory, you must stop the server using the **Stop** method and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetIdentity Method

```
BOOL SetIdentity(
   LPCTSTR lpszIdentity
);
```

Change the identity of the specified server.

Parameters

lpszIdentity

A null-terminated string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero, otherwise it will return a value of zero.

Remarks

The **SetClientIdentity** method changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **GetIdentity** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientIdentity, SetClientIdentity, GetIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetLastError Method

```
VOID SetLastError(
   DWORD dwError
);
```

Set the last error code for the specified server session.

Parameters

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR.

If the ***dwError*** parameter is specified with a value of zero, this effectively clears the error code for the last method that failed. Those methods which clear the last error code when they succeed are noted on their reference page.

Applications can retrieve the value saved by this method by calling the **GetLastError** method to determine the specific reason for failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetCommandResult, GetLastError, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetLogFile Method

```
BOOL SetLogFile(
   UINT nLogFormat,
   UINT nLogLevel,
   LPCTSTR lpszFileName
);
```

Change the current log format, level of detail and file name.

Parameters

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Value	Description
FTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of FTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A null-terminated string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the method succeeds, the return value is non-zero. If the one of the parameters are invalid, the method will return zero.

Remarks

The **SetLogFile** method can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the ***nLogFormat*** parameter with the constant FTP_LOGFILE_DELETE.

The higher the value of the ***nLogLevel*** parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record file transfers, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can

negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = FTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (pFtpServer->GetLogFile(&nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != FTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = pFtpServer->SetLogFile(nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLogFile, RenameServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetName Method

```
BOOL SetName(
   UINT nClientId,
   LPCTSTR lpszHostName
);
```

Change the host name assigned to the specified server or client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the method succeeds, the return value is non-zero. If the client ID is invalid, or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will change the host name assigned to the specified client session. If the ***nClientId*** parameter has a value of zero, the method will change default host name that was assigned to the server as part of the server configuration. If the ***nClientId*** parameter specifies a valid client session and the ***lpszHostName*** parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **GetName** method will return the host name specified by the client, and **SetName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetOptions Method

```
BOOL SetOptions(
   DWORD dwOptions
);
```

Sets the default options for this instance of the server.

Parameters

dwOptions

An unsigned integer which specifies one or more options. For a list of available options, see Server Option Constants.

Return Value

If the method is successful, it will return a non-zero value, otherwise it will return a value of zero.

Remarks

The **SetOptions** method changes the default options for the current instance of the server. This method cannot be used to change the options for an active instance of the server. If the server is active, it must be stopped before calling this method. To get the current options, use the **GetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetPriority Method

```
INT SetPriority(
   INT nPriority
);
```

Change the priority assigned to the specified server.

Parameters

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Value	Description
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **SetPriority** method can be used to change the current priority assigned to the specified server. Client connections that are accepted after this method is called will inherit the new priority as their

default priority. Previously existing client connections will not be affected by this method.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

GetPriority, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetStackSize Method

```
BOOL SetStackSize(
   DWORD dwStackSize
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change this value unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetStackSize, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetUuid Method

```
BOOL SetUuid(
   LPCTSTR lpszHostUuid
);
```

Assign a UUID to the current instance of the server.

Parameters

lpszHostUuid

A null-terminated string that specifies the server UUID, terminated with a null character. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This parameter may be NULL or point to an empty string, in which case a temporary UUID will be randomly generated for the server.

Return Value

If the method succeeds, the return value is non-zero, otherwise the method return a value of zero.

Remarks

The **SetUuid** method assigns a Universally Unique Identifier (UUID) to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. This method cannot be used to change the UUID after the server has been started. To determine the UUID assigned to an active server instance, use the **GetUuid** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetUuid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::Start Method

```
BOOL Start(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   DWORD dwOptions
);

BOOL Start(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort
);

BOOL Start(
   UINT nLocalPort
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit TLS. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see Server Option Constants. If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

In most cases, the ***lpszLocalHost*** parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and

configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this method references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

Example

```
CFtpServer *pFtpServer = new CFtpServer();

// Initialize the server configuration
pFtpServer->SetName(_T("server.company.com"));
pFtpServer->SetUuid(_T("10000000-1000-1000-1000-100000000000"));
pFtpServer->SetDirectory(_T("%ProgramData%\\MyProgram\\Server"));
pFtpServer->SetLogFile(FTP_LOGFILE_EXTENDED, 5,
_T("%ProgramData%\\MyProgram\\Server.log"));
pFtpServer->SetOptions(FTP_SERVER_LOCALUSER | FTP_SERVER_UNIXMODE);

// Start the server
pFtpServer->Start(FTP_PORT_DEFAULT);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumClients, Restart, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::Stop Method

```
BOOL Stop();
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

Restart, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::Suspend Method

```
BOOL Suspend();
```

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

Restart, Resume, Start, Stop, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::Throttle Method

```
BOOL Throttle(
   UINT nMaxClients,
   UINT nMaxClientsPerAddress,
   UINT nMaxGuests,
   DWORD dwConnectionRate
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxGuests

An integer value which specifies the maximum number of guest users that may login to the server. A value of zero disables guest logins and requires that all clients authenticate with a valid username and password. A value of -1 specifies that the maximum number of guest users should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called

lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

Restart, Resume, Start, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer Event Handlers

Method	Description
OnAuthenticate	The client requested authentication
OnCommand	The client issued a command to the server
OnConnect	The client established a connection to the server
OnDisconnect	The client has disconnected from the server
OnDownload	The client has downloaded a file from the server
OnError	The server encountered an error while handling a client request
OnExecute	The client has executed an external program on the server
OnLogin	The client has successfully authenticated the session
OnLogout	The client has logged out or reinitialized the session
OnResult	The command issued by the client has been processed by the server
OnTimeout	The client has exceeded the maximum allowed idle time
OnUpload	The client has uploaded a file to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnAuthenticate Method

```
virtual void OnAuthenticate(
   UINT nClientId,
   LPCTSTR lpszHostName,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

A virtual method that is invoked after the client has requested authentication.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A null-terminated string that specifies the host name that the client used to establish the connection.

lpszUserName

A null-terminated string that specifies the user name provided by the client.

lpszPassword

A null-terminated string that specifies the cleartext user password provided by the client.

Return Value

None.

Remarks

The **OnAuthenticate** event handler is invoked when the client has requested authentication by sending the USER and PASS command to the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The event handler can call the **AuthenticateClient** method to authenticate the client session. To reject an authentication attempt because of an invalid user name or password, the handler should call the **SendResponse** method and specify a result code of 430.

If the client session is not authenticated, the server will perform its default authentication process. If the FTP_SERVER_ANONYMOUS configuration option has been specified, and the client has logged in as an anonymous user, the session will be authenticated as a restricted user. If the FTP_SERVER_LOCALUSER configuration option has been specified, the user name and password is checked against the local user database. If the credentials are valid, the session will be authenticated as a regular user. If neither the FTP_SERVER_ANONYMOUS or FTP_SERVER_LOCALUSER options were specified, the default action is to reject all authentication attempts.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SendResponse, OnLogin, OnLogout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnCommand Method

```
virtual void OnCommand(
   UINT nClientId,
   LPCTSTR lpszCommand
);
```

A virtual method that is invoked after the client has sent a command to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see Server Commands.

Return Value

None.

Remarks

The **OnCommand** event handler is invoked after the client has sent a command to the server, but before the command has been processed. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event handler is invoked for all commands issued by the client, including invalid or disabled commands. If the event handler processes the command, it must call the **SendResponse** method to send a success or error response back to the client. If this is not done, the server will perform the default processing for the command.

Although this event handler will provide the command name, the full command line can be obtained by using the **GetCommandLine** method. Individual command parameters can be obtained by using the **GetCommandParam** and **GetCommandParamCount** methods.

It is not necessary to use this event handler to disable a command. The **EnableCommand** method can be used to enable or disable specific commands, and the **IsCommandEnabled** method can be used to determine if a command is enabled.

If this event handler is used to implement a custom command, it is recommended that you use the **IsClientAuthenticated** method to determine whether or not the client session has been authenticated. Unless there is a specific need for the custom command to be used before a client has logged in, the application should not take any action and send a 530 result code back to the client indicating authentication is required.

Example

```
VOID CMyFtpServer::OnCommand(UINT nClientId, LPCTSTR lpszCommand)
{
    // Implement a custom command named TIME that will return the local time
    if (lstrcmp(lpszCommand, _T("TIME")) == 0)
    {
        // The command should not have any parameters
        if (GetCommandParamCount(nClientId) > 0)
        {
```

```
            SendResponse(nClientId, FTP_REPLY_BADARG);
            return;
        }

        if (IsClientAuthenticated(nClientId))
        {
            CString strTime = CTime::GetCurrentTime().Format(_T("%Y-%m-%d
%H:%M:%S"));
            SendResponse(nClientId, FTP_REPLY_CMDOK, strTime);
        }
        else
        {
            // The client has not logged in, return an error
            SendResponse(nClientId, FTP_REPLY_NOLOGIN);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisconnectClient, EnableCommand, IsCommandEnabled, OnDisconnect, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnConnect Method

```
virtual void OnConnect(
   UINT nClientId,
   LPCTSTR lpszAddress
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A null-terminated string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and the TLS handshake has been performed, if security has been enabled. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will terminate the client session prior to this event handler being invoked.

If this event handler is not implemented, the server will perform the default action of accepting the connection and sending a standard greeting to the client. If you want your application to send a custom greeting to the client when it connects, call the **SendResponse** method, specifying a result code of 220 and a message of your choice.

To reject a connection, call the **SendResponse** method to send an error response to the client. Typically the result code value would be 421 to indicate that the server will not accept the connection. Next, call the **DisconnectClient** method to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisconnectClient, OnDisconnect, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnDisconnect Method

```
virtual void OnDisconnect(
   UINT nClientId
);
```

A virtual method that is invoked immediately before the client is disconnected from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked immediately before the client is disconnected from the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

OnConnect, OnLogout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnDownload Method

```
virtual void OnDownload(
   UINT nClientId,
   LPCTSTR lpszFileName,
   DWORD dwTimeElapsed,
   ULARGE_INTEGER uiBytesCopied
);
```

A virtual method that is invoked after the client has successfully downloaded a file from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the local path name of the file that was downloaded from the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElased

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was downloaded by the client.

Return Value

None.

Remarks

The **OnDownload** event handler is invoked after the client has successfully downloaded a file from the server using the RETR command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OnCommand, OnUpload

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnError Method

```
virtual void OnConnect(
   UINT nClientId,
   UINT nEventId,
   DWORD dwError
);
```

A virtual method that is invoked when the server encounters an error while handling a client request.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see Server Event Constants.

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit TLS but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- A network error occurs when attempting to process a command issued by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

OnConnect, OnCommand

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnExecute Method

```
virtual void OnExecute(
   UINT nClientId,
   LPCTSTR lpszProgram,
   DWORD dwExitCode
);
```

A virtual method that is invoked after the client has successfully executed an external program on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgram

A null-terminated string that specifies the name of the program that was executed. This is the registered program name, and not a full path to the executable and its command arguments.

dwExitCode

An unsigned integer that specifies the exit code that was returned by the program.

Return Value

None.

Remarks

The **OnExecute** event handler is invoked after the client has successfully executed an external program using the SITE EXEC command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event will only be generated if the client has the FTP_ACCESS_EXECUTE permission. Clients are not granted this permission by default, and must be explicitly permitted to execute external programs. If the client does have this permission, it can only execute specific programs that have been registered by the server application using the **RegisterProgram** method.

The **GetProgramOutput** method can be used to obtain the unfiltered output from the external command, while the **GetProgramText** method will return filtered output from the program that contains only printable text characters.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramOutput, GetProgramText, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnLogin Method

```
virtual void OnLogin(
   UINT nClientId,
   LPCTSTR lpszUserName,
   LPCTSTR lpszDirectory,
   LPCTSTR dwUserAccess
);
```

A virtual method that is invoked after the client has successfully authenticated the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A null-terminated string that specifies the user name.

lpszDirectory

A null-terminated string that specifies the full path to the home directory for the client. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwUserAccess

An unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see User Access Constants.

Return Value

None.

Remarks

The **OnLogin** event handler is invoked after the client has successfully authenticated itself using the USER and PASS commands. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

To convert the home directory path to a virtual path name that is relative to the server root directory, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualUser, AuthenticateClient, OnAuthenticate, OnLogout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnLogout Method

```
virtual void OnLogout(
   UINT nClientId,
   LPCTSTR lpszUserName
);
```

A virtual method that is invoked after the client has logged out or reinitialized the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A null-terminated string that specifies the user name.

Return Value

None.

Remarks

The **OnLogout** event handler is invoked after the client has successfully logged out using the QUIT command or reinitialized the session using the REIN command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The application should not depend on this event handler always being invoked when a client is disconnected from the server. This event only occurs when the client sends the QUIT or REIN commands and will not be invoked if the client connection is aborted or disconnected for some other reason, such as exceeding the idle timeout period. If the application needs to update data structures or perform some cleanup when a client disconnects, that should be done in the **OnDisconnect** event handler.

The application should not call the **DisconnectClient** method in the handler for this event because the client is either in the process of disconnecting or expects that it can submit new credentials to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OnConnect, OnDisconnect, OnLogin

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnResult Method

```
virtual void OnResult(
   UINT nClientId,
   LPCTSTR lpszCommand,
   UINT nResultCode
);
```

A virtual method that is invoked after the server has processed a command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string that specifies the command that was issued by the client. This value contains only the command and not the additional parameters that may have been specified.

nResultCode

An integer value that specifies the result code that was sent to the client.

Return Value

None.

Remarks

The **OnResult** event handler is invoked after the server has processed a command issued by the client. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

To obtain the complete command line, use the **GetCommandLine** method, or use the **GetCommandParam** method to get the value of specific command parameters. If the application requires the text message that was sent to the client along with the result code, use the **GetCommandResult** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandLine, GetCommandParam, GetCommandResult, OnCommand

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnTimeout Method

```
virtual void OnTimeout(
   UINT nClientId,
   UINT nIdleTime,
   UINT nElapsed
);
```

A virtual method that is invoked after the client has exceeded the maximum allowed idle time.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nIdleTime

An unsigned integer value that specifies the current idle timeout period for the client in seconds.

nElapsed

An unsigned integer value that specifies the number of seconds that the client has been idle.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked after the client has has exceeded the maximum allowed idle time. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event handler will be invoked prior to the client being disconnected from the server. This event will never occur during a file transfer or directory listing. The **SetClientIdleTime** method can be used to change or refresh the idle timeout period for the session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

GetClientIdleTime, OnCommand, OnResult, SetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnUpload Method

```
virtual void OnUpload(
   UINT nClientId,
   LPCTSTR lpszFileName,
   DWORD dwTimeElapsed,
   ULARGE_INTEGER uiBytesCopied
);
```

A virtual method that is invoked after the client has successfully uploaded a file to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the local path name of the file that was uploaded to the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElased

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was uploaded by the client.

Return Value

None.

Remarks

The **OnUpload** event handler is invoked after the client has successfully uploaded a file to the server using the APPE, STOR or STOU command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OnCommand, OnDownload

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CFtpServer Data Structures

- FTPCLIENTCREDENTIALS
- FTPSERVERTRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPCLIENTCREDENTIALS

The **FTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _FTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    TCHAR szHostName[FTP_MAXHOSTNAME];
    TCHAR szUserName[FTP_MAXUSERNAME];
    TCHAR szPassword[FTP_MAXPASSWORD];
} FTPCLIENTCREDENTIALS, *LPFTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value reserved for future use. This member will always be initialized to a value of zero.

szHostName

A null-terminated string that specifies the server host name.

szUserName

A null-terminated string that specifies the user name.

szPassword

A null-terminated string that specifies the user password.

Remarks

When an instance of this structure is passed to the **GetClientCredentials** method, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPSERVERTRANSFER

The **FTPSERVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _FTPSERVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} FTPSERVERTRANSFER, *LPFTPSERVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Value	Description
FTP_FILE_READ	The file was opened for reading. This mode indicates that the client issued the RETR command to download the contents of a file from the server to the client system. The ***szFileName*** member specifies the name of the local file on the server that was downloaded by the client.
FTP_FILE_WRITE	The file was opened for writing. This mode indicates that the client issued the STOR or STOU command to upload the contents of a file from the client system to server. The ***szFileName*** member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.
FTP_FILE_APPEND	The file was opened for writing. This mode indicates the client issued the APPE command to upload the contents of a file from the client system and append the data to a file on the server. If the file did not exist, then it was created. The ***szFileName*** member specifies the name of the local file that was appended to or created by the client.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a ULARGE_INTEGER union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the

Windows SDK documentation. The application should not make the assumption that this is the actual size of the file. If the client specified a restart offset using the REST command, this value would only represent the number of bytes transferred from that byte offset, not the total file size.

szFileName

A null-terminated string value that will contain the full path to the local file that was transferred. The ***dwFileAccess*** member determines whether the file name represents a file that was downloaded by the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **GetTransferInfo** method, the ***dwSize*** member must be initialized to the size of the structure, otherwise the method will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

GetTransferInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Class Library

Transfer files between the local system and a web server, execute scripts and perform remote file management functions.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CHttpClient
File Name	CSHTPV12.DLL
Version	12.0.3024.1286
LibID	E5D90316-2B5F-4E36-BD24-F58F1B47C3BF
Import Library	CSHTPV12.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 7230, RFC 7540

Overview

The Hypertext Transfer Protocol (HTTP) is a lightweight, stateless application protocol that is used to access resources on web servers, as well as send data to those servers for processing. The library provides direct, low-level access to the server and the commands that are used to retrieve resources (i.e.: documents, images, etc.). The library also provides a simple interface for downloading resources to the local host, similar to how the FTP library can be used to download files.

In a typical session, you would use the class to establish a connection, send a request (to download a resource, post data for processing, etc.), read the data returned by the server and then disconnect. It is the responsibility of the client to process the data returned by the server, depending on the type of resource that was requested.

This class supports HTTP versions 1.0, 1.1, 2.0 and 3.0. Secure connections use TLS 1.2 or later and supports the use of client certificates if required. There are several methods designed to make it easy to submit JSON and XML requests and include custom headers which may be required by REST APIs. The class also supports standard authentication methods and the use of OAuth bearer tokens.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1

(SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Class Methods

Class	Description
CHttpClient	Constructor which initializes the current instance of the class.
~CHttpClient	Destructor which releases resources allocated by the class.
Method	Description
AddField	Append the form field and its value to the current form.
AddFile	Append the contents of the file to the current form.
AddHeaders	Add one or more request headers for the current client session.
AttachHandle	Attach the specified client handle to this instance of the class.
AttachThread	Attach the specified client handle to another thread.
Authenticate	Specify authentication information for restricted resources.
Cancel	Cancel the current blocking operation.
ClearForm	Remove all defined fields from the current form.
ClearHeaders	Resets the current request and response headers to their default values.
CloseFile	Close the file opened on the server.
Command	Send a command to the server.
Connect	Connect to the specified server.
ConnectUrl	Establish a client connection using the specified URL.
CreateFile	Create or replace a file on the server.
CreateForm	Create a new form to submit to the server.
CreateSecurityCredentials	Create a new security credentials structure.
DeleteFile	Remove a file from the server.
DeleteField	Delete the form field and its value from the current form.
DeleteHeaders	Delete all of the response or request headers for the current session.
DeleteSecurityCredentials	Delete a previously created security credentials structure.
DestroyForm	Destroy the current form and free the memory allocated for it.
DetachHandle	Detach the handle for the current instance of this class.
DisableEvents	Disable asynchronous event notification.
DisableTrace	Disable logging of network function calls.
Disconnect	Disconnect from the current server.
DownloadFile	Download a file from the server to the local system.
EnableCompression	Enable or disable support for data compression.
EnableEvents	Enable asynchronous event notification.
EnableTrace	Enable logging of network function calls to a file.

FreezeEvents	Suspend asynchronous event processing.
GetBearerToken	Return the current OAuth 2.0 bearer token for the client session.
GetContentLength	Return the size of the current resource in bytes.
GetContentType	Return the content type for the current resource.
GetCookie	Return information about the specified cookie.
GetData	Copy the specified resource to a local buffer.
GetDefaultUserAgent	Return the default user agent string included with requests.
GetDefaultZoneId	Return the default security zone for downloaded files.
GetEncodingType	Determines which content encoding option is enabled.
GetErrorString	Return a description for the specified error code.
GetFile	Copy a file from the server to the local system.
GetFilePathUrl	Return a normalized URL for the specified file on the server.
GetFileSize	Return the size of a file on the server.
GetFileTime	Return the date and time a file on the server was last modified.
GetFileZoneInfo	Return file zone identifier information for the specified local file.
GetFirstCookie	Return the first cookie set by the server.
GetFirstHeader	Return the name and value of the first request or response header field.
GetFormProperties	Return the properties of the specified form.
GetHandle	Return the client handle used by this instance of the class.
GetHeader	Return the value of the specified response header field.
GetLastError	Return the last error code.
GetNextCookie	Return the next cookie set by the server.
GetNextHeader	Return the name and value of the next request or response header field.
GetOption	Return the enabled/disabled state of a specified option.
GetPriority	Return the current priority for file transfers.
GetRedirectUrl	Return the URL for a redirected resource.
GetResourceUrl	Return the URL for the current resource.
GetResultCode	Return the result code from the previous command.
GetResultString	Return the result string from the previous command.
GetSecurityInformation	Return security information about the current client connection.
GetSecurityZoneId	Returns the security zone identifier for the specified resource.
GetStatus	Return the current client status.
GetText	Download the contents of a text file or resource to a string buffer.
GetTimeout	Return the number of seconds until an operation times out.

GetTransferStatus	Return data transfer statistics.
HttpEventProc	Callback method that processes events generated by the client.
IsBlocking	Determine if the client is blocked, waiting for information.
IsConnected	Determine if the client is connected to the server.
IsInitialized	Determine if the class has been successfully initialized.
IsReadable	Determine if data can be read from the server.
IsWritable	Determine if data can be written to the server.
NormalizePath	Normalize the specified resource returning a canonical path.
NormalizeUrl	Normalize the specified URL returning a canonical path.
OpenFile	Open a file on the server for reading.
PatchData	Submits JSON or XML patch data to the server and returns the response.
PostData	Post data from a local buffer to the server and returns the response.
PostFile	Submit the contents of a local file to the server.
PostJson	Submit JSON structured data to the server and returns the response.
PostText	Submit plain text data to the server and returns the response.
PostXml	Submit XML structured data to the server and returns the response.
ProxyConnect	Establish a connection with the specified proxy server.
PutData	Create a file on the server using the contents of a local buffer.
PutFile	Copy a file from the local system to the server.
PutText	Create a text file on the server from the contents of a string buffer.
Read	Read data from the server.
RegisterEvent	Register an event callback function.
SetBearerToken	Set the value of the OAuth 2.0 bearer token for the client session.
SetContentType	Set the content type for the request.
SetCookie	Set the value of the specified cookie.
SetDefaultUserAgent	Set the default user agent string included with requests.
SetDefaultZoneId	Set the default security zone for downloaded files.
SetEncodingType	Specifies the type of encoding to be applied to data submitted to the server.
SetFileZoneInfo	Set or remove the zone indentifier associated with the specified file.
SetFormProperties	Modify the properties of the current form.
SetHeader	Set the value of a request header field.
SetLastError	Set the last error code.
SetOption	Enable or disable the specified option.
SetPriority	Set the priority for file transfers.
SetTimeout	Set the number of seconds until an operation times out.

ShowError	Display a message box with a description of the specified error.
SubmitForm	Submit the current form to the server for processing.
SubmitRequest	Submit a request to the server and return the response.
UploadFile	Upload a file from the local system to the server.
ValidateHostName	Validate the specified host name and return the resolved IP address.
ValidateUrl	Check the contents of a string to ensure it represents a valid URL.
VerifyFile	Compare the size of a local file against a file stored on the server.
Write	Write data to the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::~CHttpClient

```
~CHttpClient();
```

The **CHttpClient** destructor releases resources allocated by the current instance of the **CHttpClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CHttpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CHttpClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::AddField Method

```
INT AddField(
   LPCTSTR lpszFieldName,
   LPVOID lpFieldData,
   DWORD cbFieldData
);

INT AddField(
   LPCTSTR lpszFieldName,
   LPCTSTR lpszFieldData
);
```

The **AddField** method adds a new field to the current form.

Parameters

lpszFieldName

A null-terminated string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpFieldData

The form field data. Typically this will either by a pointer to an array of bytes or a string which specifies the value for the form field. If no data is to be associated with the form field, then this argument may be NULL.

cbFieldData

An unsigned integer value which specifies the number of bytes of data in the form field. If this value is 0xFFFFFFFF (-1) then it is assumed that the ***lpFieldData*** parameter is a pointer to a null-terminated string. If ***lpFieldData*** is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **AddField** method is used to add a field and its associated value to a form created using the **CreateForm** method. If the field name has already been added to the form, the previous value is deleted and replaced by the new value.

If the Unicode version of this method is called and the ***cbFieldData*** parameter is -1, the ***lpFieldData*** parameter must point to a wide string terminated with a null character. The string will be automatically converted to UTF-8 when the form data is submitted to the server.

Example

```
CString strResult;
INT nResult = 0;

pClient->CreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);
pClient->AddField(_T("UserName"), lpszUserName);
pClient->AddField(_T("Password"), lpszPassword);

nResult = pClient->SubmitForm(strResult);
pClient->DestroyForm();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, ClearForm, CreateForm, DeleteField, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::AddFile Method

```
INT AddFile(
   LPCTSTR lpszFieldName,
   LPCTSTR lpszFileName
);
```

The **AddFile** method adds the contents of a file to the specified form.

Parameters

lpszFieldName

A null-terminated string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpszFileName

A null-terminated string which specifies the name of the file. The contents of the file will be added to the form data that is submitted to the server.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddField, ClearForm, CreateForm, DeleteField, PostFile, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::AddHeaders Method

```
BOOL AddHeaders(
   LPCTSTR lpszHeaderList
);
```

The **AddHeaders** method adds one or more request headers for the current client session.

Parameters

lpszHeaderList

A null-terminated string which specifies one or more header values which should be set for the current client session. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If a connection has not been established with a server, or the header list contains invalid values, this method returns a value of zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AddHeaders** method enables your application to set one or more header values by providing a list of name/value pairs separated by a colon. Multiple header values may be provided by separating them with a newline character. This method is similar to calling the **SetHeader** method for each value. When the list of header values is parsed, extraneous whitespace is ignored; however, if the header list contains invalid text (for example, a missing colon separating a header name from its value) the method will fail and an error will be returned.

Exercise caution when providing a header list created directly from user input, such as a list of values input using a textbox control. Any header values which have been previously set by your application can be overridden by this method and may yield unpredictable results. If the service you are using requires a custom authorization header, such as an API token or other user credentials, allowing users to directly modify request header values this way can present a security risk.

Example

```
// Define a list of header values which should be included with the
// request submitted to the server
LPCTSTR lpszHeaderList = _T("X-API-Token: 99d2fe39-0246-4efa-98e0-4d775579fa5d\n")
\
                         _T("X-API-UserId: someuser@domain.tld\n") \
                         _T("X-API-Version: 1.1\n")

if (!pClient->AddHeaders(lpszHeaderList))
{
    // Unable to add the headers for this client session
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Authenticate Method

```
INT Authenticate(
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);

INT Authenticate(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

Parameters

nAuthType

An unsigned integer value which specifies the method to be used when authenticating the client. If this argument is not specified, then basic authentication will be used. The following values are recognized.

Value	Description
HTTP_AUTH_NONE	No client authentication should be performed. The *lpszUserName* and *lpszPassword* parameters are ignored and current authentication settings are cleared.
HTTP_AUTH_BASIC	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the *lpszPassword* parameter must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error. This function will not automatically refresh an expired token.

lpszUserName

A null-terminated string that specifies the username used to authenticate the client session. This parameter may be NULL or an empty string if a user name is not required for the specified authentication type.

lpszPassword

A null-terminated string that specifies the password used to authenticate the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will set the `Authorization` request header for the client session using the credentials

provided by the caller. This method will always override any custom **Authorization** header value that may have been previously set using the **SetRequestHeader** method.

If both the *lpszUserName* and *lpszPassword* parameters are NULL pointers or specify zero-length strings, the current authentication type will always be set to HTTP_AUTH_NONE regardless of the value of the *nAuthType* parameter. This effectively clears the current user credentials for the client session.

If the web service requires OAuth 2.0 authentication, it is recommended you use the **SetBearerToken** method to specify the access token.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetBearerToken, SetBearerToken, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::CHttpClient

```
CHttpClient();
```

The **CHttpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **HttpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

The instance of this class is initially attached to the thread which created it and only the owning thread may invoke methods in that instance. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

To enable any thread to invoke class methods regardless of ownership, specify the HTTP_OPTION_FREETHREAD option when calling the **Connect** or **ConnectUrl** method. It is important to note this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the instance is synchronized. If one thread invokes a method, it must ensure no other thread will invoke another method at the same time for the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

~CHttpClient, AttachHandle, AttachThread, Connect, ConnectUrl, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ClearForm Method

```
VOID ClearForm();
```

The **ClearForm** method clears the current form, removing all fields.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AddField, AddFile, CreateForm, DeleteField, DestroyForm, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ClearHeaders Method

```
VOID ClearHeaders();
```

The **ClearHeaders** method resets the current request and response headers to their default values.

Parameters

None.

Return Value

None.

Remarks

The **ClearHeaders** method removes all custom header values for the current client session and clears the response headers for the previous request. This method is useful when using persistent connections and you want to ensure that any custom header values from the previous request will not be included with subsequent requests made during the same session. Because this method clears all of the request headers for the client session, this means it will also remove any cookies which have been set using the **SetCookie** method.

This method will clear any authentication credentials or tokens specified for the current client session. If the next request you make requires authentication, you must provide those credentials again using the **Authenticate** or **SetBearerToken** method.

It is not necessary to use this method if the **HTTP_OPTION_KEEPALIVE** option was not specified when the connection was established. The headers for the previous request are automatically cleared when a new connection is established.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Authenticate, Connect, GetCookie, GetHeader, SetBearerToken, SetCookie, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::CloseFile Method

```
INT CloseFile();
```

The **CloseFile** method flushes the internal client buffers and closes the previously opened file on the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the file is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the method returns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Command, OpenFile, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Command Method

```
INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszResource,
   LPBYTE lpParameter,
   DWORD cbParameter
);

INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszResource,
   LPCTSTR lpszParameter
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for extended commands not directly supported by the API.

Parameters

lpszCommand

A null-terminated string which specifies the command to be executed by the server. The following table lists the standard commands recognized by most HTTP servers. Other commands may also be used, such as those extensions used by WebDAV to edit and manage files on a server.

Command	Description
GET	Return the contents of the specified resource. This command is recognized by all servers.
HEAD	Return only header information for the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
POST	Post data to the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
PUT	Create or replace the specified resource on the server. This command is recognized by servers that support at least version 1.0 of the protocol. Not all servers support this command.
DELETE	Delete the specified resource from the server. This command is recognized by servers that support at least version 1.1 of the protocol. Not all servers support this command.

lpszResource

A null-terminated string which specifies the resource to be used with the command. This can be the name of a file, an executable script or any other valid resource name recognized by the server. Resource names must be absolute and include the complete path to the resource.

lpParameter

A pointer to a byte array which contains data that is to be passed to the command as one or more parameters. Typically this is used to pass additional information to a script that executes on the server. The data is encoded according to the encoding type specified for the client session. If the resource does not require any parameters, this value should be NULL.

cbParameter

Specifies the number of bytes stored in the parameter buffer. If the resource does not require any parameters, this value should be zero.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Not all servers support all of the listed commands, and some commands may require specific changes to the server configuration. In particular, the PUT and DELETE commands typically require that configuration changes be made by the site administrator. All servers will support the use of the GET command, and all servers that support at least version 1.0 of the protocol will support the POST command.

If the *lpszResource* parameter specifies a path that contains reserved or restricted characters, such as a space, it will automatically be URL encoded by the library.

It is permissible to include a query string in the resource name specified by the *lpszResource* parameter. Query strings begin with a question mark, and then are followed by one or more name/value pairs separated by an equal sign. For example, the following resource includes a query string:

```
/cgi-bin/test.cgi?field1=value1&field2=value2
```

In this case, the query string is "?field1=value1&field2=value2". If the query string contains reserved or restricted characters, such as spaces, then it will be automatically URL encoded prior to being sent to the server. If additional resource data is specified in the *lpParameter* argument along with a query string in the resource name, the action taken by the library depends on the command being sent. If the command is a POST or PUT command, then query string is included with command request to the server and the parameter data is sent separately. For example, if the POST command was used, the script running on the server would see that both query data and form data has been provided to it. However, if any other command is specified, the parameter data is simply appended to the query string.

The *lpParameter* argument is used to pass additional information to the server when a resource is requested. This is most commonly used to provide information to scripts, similar to how arguments are used when executing a program from the command line. Unless the POST command is being executed, the data in the buffer will automatically be encoded using the current encoding mechanism specified for the client. By default, the data is URL encoded, which means that any spaces and non-printable characters are converted to printable characters before submitted to the server. The type of encoding that is performed can be set by calling the **SetEncodingType** method. Although the default encoding is appropriate for most applications, those that submit XML formatted data may need to change the encoding type.

Only one request may be in progress at one time for each client session. Use the **CloseFile** method to terminate the request after all of the data has been read from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion
);
```

The **Connect** method is used to establish a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default

	proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects

	connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the client will default to using TLS 1.2 or later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create a version value and the following values are defined:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.

HTTP_VERSION_09	The client should use the original one-line protocol which includes no version number and no request header block. This version has been deprecated and should only be used with legacy servers which do not support the protocol standard. This version of the protocol does not support virtual hosts and is not supported by most modern web services.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.
HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. If the client or server does not support HTTP/2, the client will automatically attempt to use an earlier version of the protocol.
HTTP_VERSION_30	The client should use the HTTP/3 protocol standard defined in RFC 9114. This protocol version is similar to earlier versions of the protocol in terms the request methods and server responses, but the underlying transport mechanism is significantly different. Instead of using TCP, this version of the protocol uses multiplexed UDP connections designed to reduce the overhead of TCP while maintaining reliability. If the client or server does not support HTTP/3, the client will automatically attempt to use an earlier version of the protocol.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Most applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Under some circumstances, a server may forcibly close or abort a connection if the request fails. For example, this can happen when a malformed request is submitted to the server or a server-side error occurs. When this happens, the application will need to disconnect from the server and establish a new connection, even if the keep-alive option has been specified.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the method will fail and return FALSE.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to invoke methods in the class instance, regardless of which thread created it. It is important to note that this option disables certain internal safety checks may result in unexpected behavior unless access to the class instance is synchronized.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, Disconnect, IsConnected, ProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ConnectUrl Method

```
BOOL ConnectUrl(
   LPCTSTR lpszUrl,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **ConnectUrl** method establishes a connection with the specified server using a URL.

Parameters

lpszUrl

A null-terminated string which specifies the URL for the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the

	content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the client will default to using TLS 1.2 or later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.

HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ConnectUrl** method uses an HTTP URL to establish a connection with a server. The URL must be in the following format:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified. The **ValidateUrl** method can be used to verify that a URL is valid prior to calling this method.

The **ConnectUrl** method is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection methods. If you only need to upload or download a file using a URL, then refer to the **UploadFile** and **DownloadFile** methods.

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **ConnectUrl** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Most applications should use HTTP_VERSION_DEFAULT as the value for the *dwVersion* parameter. This will default to an appropriate version for the Windows platform and ensures the broadest

compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

If your application specifies the HTTP_OPTION_HTTP2 option, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of Windows do not support the features required for a secure HTTP/2 connection. If the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the method will fail and return FALSE.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to invoke methods in the class instance, regardless of which thread created it. It is important to note that this option disables certain internal safety checks may result in unexpected behavior unless access to the class instance is synchronized.

Example

```
CHttpClient httpClient;

if (!httpClient.ConnectUrl(_T("https://example.com/")))
{
    httpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect, DownloadFile, UploadFile, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::CreateFile Method

```
INT CreateFile(
   LPCTSTR lpszRemoteFile,
   DWORD dwFileLength
);
```

The **CreateFile** method creates the specified file on the server.

Parameters

lpszRemoteFile

Points to a string that specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

dwFileLength

Specifies the length of the file that will be created on the server. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CreateFile** method uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this method is successful, the client should then use the **Write** method to send the contents of the file to the server. Once all of the data has been written, the **CloseFile** method should be called to close the file and complete the operation. Note that this method is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **Write** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file in a single method call, use the **PutData** method instead of calling **CreateFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFile, OpenFile, PutData, PutFile, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::CreateForm Method

```
BOOL CreateForm(
   LPCTSTR lpszAction,
   UINT nFormMethod,
   UINT nFormType
);
```

The **CreateForm** method creates a new form for use with the other form-related functions.

Parameters

lpszAction

A null-terminated string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. This parameter may be one of the following values:

Value	Description
HTTP_METHOD_DEFAULT	The form data should be submitted using the default method, using the GET command.
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. This parameter may be one of the following values:

Value	Description
HTTP_FORM_DEFAULT	The form data should be submitted using the default encoding method.
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CreateForm** method is used to create a new form that will be populated with values and then submitted to the server for processing. When the form is no longer needed, it should be destroyed using the **DestroyForm** function.

Example

```
CString strResult;
INT nResult = 0;

pClient->CreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);
pClient->AddField(_T("UserName"), lpszUserName);
pClient->AddField(_T("Password"), lpszPassword);

nResult = pClient->SubmitForm(strResult);
pClient->DestroyForm();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddField, AddFile, ClearForm, DeleteField, DestroyForm, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              HTTP_PORT_SECURE,
                              HTTP_TIMEOUT,
                              HTTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DeleteField Method

```
INT DeleteField(
   LPCTSTR lpszFieldName
);
```

The **DeleteField** method deletes the specified field from the current form.

Parameters

lpszFieldName

Points to a string that specifies the name of the field to delete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddField, AddFile, ClearForm, CreateForm, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DeleteFile Method

```
INT DeleteFile(
   LPCTSTR lpszFileName
);
```

The **DeleteFile** method deletes the specified file from the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the DELETE command to delete the specified file from the server. The server must be configured to support this command, and client must have the appropriate permission to delete the file, or an error will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DeleteHeaders Method

```
BOOL DeleteHeaders(
   UINT nHeaderType
);
```

Delete all of the response or request headers for the current session.

Parameters

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that have been set for the next request sent to the server. This will clear all of the header values that were set using the **SetRequestHeader** method.
HTTP_HEADERS_RESPONSE	Delete all of the headers that were set in response to the previous request. A call to the **GetResponseHeader** method to obtain a specific header value will fail after this method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DeleteHeaders** method will release the memory allocated for the request or response headers for the current session. This method is typically used to clear all of the current request headers so that the application may create a new set of headers when a persistent connection is being used. The memory allocated for the request and response headers is normally released when the session handle is closed by calling the **Disconnect** method.

Whenever a request for a resource is sent to the server, the response headers from the previous request are automatically cleared. It is not necessary for an application to call the **DeleteHeaders** method to delete the response headers prior to each request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetFirstHeader, GetNextHeader, GetHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DestroyForm Method

```
VOID DestroyForm();
```

The **DestroyForm** method destroys the current form, releasing the memory allocated for it.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AddField, AddFile, ClearForm, CreateForm, DeleteField, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Connect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::DownloadFile Method

```
BOOL DownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPHTTPTRANSFERSTATUS lpStatus
   HTTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **DownloadFile** method downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.

HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **EventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DownloadFile** method provides a convenient way for an application to download a file in a single method call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then download the file to the local system. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the method will fail and return a value of zero.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **DownloadFile** method is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **GetFile** to download the file.

Example

```
CHttpClient httpClient;
CString strLocalFile = _T("c:\\temp\\database.mdb");
CString strFileURL = _T("http://www.example.com/updates/database.mdb");

// Download the file using the specified URL
if (!httpClient.DownloadFile(strLocalFile, strFileURL))
{
    httpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetFileZoneInfo, GetTransferStatus, HttpEventProc, SetFileZoneInfo, UploadFile, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::EnableCompression Method

```
INT EnableCompression(
   BOOL bEnable
);
```

The **EnableCompression** method enables or disables support for data compression.

Parameters

bEnable

An optional boolean value which specifies if data compression should be enabled or disabled. A non-zero value enables compression, while a value of zero will disable compression. If this parameter is omitted, compression will be enabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpEnableCompression** method is used to indicate to the server whether or not it is acceptable to compress the data that is returned to the client. If compression is enabled, the client will advertise that it will accept compressed data by setting the **Accept-Encoding** request header. The server will decide whether a resource being requested can be compressed. If the data is compressed, the library will automatically expand the data before returning it to the caller.

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. Disabling compression informs the server that the client is not willing to accept compressed data; this is the default.

If the **SetHeader** method is used to explicitly set the **Accept-Encoding** header to request compressed data and compression is not enabled, the library will not attempt to automatically expand the data returned by the server. In this case, the raw compressed data will be returned to the caller and the application is responsible for processing it. This behavior is by design to maintain backwards compatibility with previous versions of the library that did not have internal support for compression.

To determine if the server compressed the data returned to the client, use the **GetHeader** method to get the value of the **Content-Encoding** header. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

Enabling compression is only meaningful when downloading files from a server that supports file compression. It has no effect on file uploads.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetData, GetFile, GetHeader, GetText, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This

	event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

When logging is enabled for HTTP/2 and HTTP/3 client sessions, it is normal to see HTTP/1.1 in the

request and server response. They are binary protocols and the request and response header blocks emulate a standard HTTP/1.1 text response for backwards compatibility. Applications running on the server should work in the same way regardless of which protocol version is selected, however it is possible to check the server environment to determine which version of HTTP was used with the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetBearerToken Method

```
INT GetBearerToken(
   LPTSTR lpszBearerToken,
   INT nMaxLength
);

INT GetBearerToken(
   CString& strBearerToken
);
```

The **GetBearerToken** method returns the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

lpszBearerToken

A null-terminated string buffer which will contain the bearer token when the method returns. The string will be null-terminated and the buffer must be large enough to accommodate the entire bearer token or the method will fail. This parameter cannot be a NULL pointer. An alternate version of this method accepts a **CString** object.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the length of the bearer token string. A return value of zero indicates that no bearer token has been specified. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns the bearer token which was previously set by a call to **SetBearerToken**. Bearer tokens can be very long strings and do not contain any human readable information. For Google services, these tokens are usually about 180 characters in length. For Microsoft services, their bearer tokens are typically 1,200 characters in length. It is recommended that you provide a buffer size of at least 2,000 characters.

Your application should not store the bearer tokens provided by a web service. These tokens are short-lived and typically only valid for about an hour. If the token has expired, the authorization to access the resource will fail and it must be refreshed. The refresh tokens used to acquire a new bearer token should be stored and they are typically valid for a period of months.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Authenticate, SetBearerToken

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetContentLength Method

```
BOOL GetContentLength(
   LPDWORD lpLength
);

BOOL GetContentLength(
   ULARGE_INTEGER *lpLength
);
```

The **GetContentLength** method returns the size of the current resource in bytes.

Parameters

hClient

Handle to the client session.

lpLength

A pointer to a unsigned integer which will contain the content length for the current resource. This value will be initialized to zero and updated with the length when the method returns. If it's possible the server will return more than 4GiB of data to the client, this parameter should point to an unsigned 64-bit integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When this method is called after a GET or POST request has completed, the ***lpLength*** parameter will contain the total number of bytes returned by the server. If this method is called while a request is being processed (for example, from within an event handler), it will return the value of the Content-Length response header provided by the server.

Most servers will only include a Content-Length header in the response for static resources like documents or image files. For other types of resources where the response payload is generated dynamically, such as server-side scripts, servers typically use "chunked" output and there is no Content-Length header. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. In this case, the content length can only be determined after the request has completed.

This method should only be used after a request has completed and will not return a value for dynamic resources as they are being retrieved from the server. To obtain the current number of bytes copied from within an event handler, use the **GetTransferStatus** method.

If the content type for the current resource is textual, the content length reported by the server may be different than what is stored on the local system due to different end-of-line character conventions. For example, on Windows the end of a line of text is indicated by a carriage-return and linefeed pair, while on Linux the end-of-line is indicated by a single linefeed. Depending on the options specified when the request is made, text returned by the server may be normalized for the Windows platform.

This method does not use the HEAD command to request the size of a document on the server. It will only return the content length of a resource which has already been requested, typically after using methods such as **GetText** or **PostXml**. To ask the server for the size of document without downloading the contents of that document, use the **GetFileSize** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetContentType, GetFileSize, GetHeader, GetTransferStatus, SetContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetContentType Method

```
INT GetContentType(
   LPTSTR lpszContentType,
   INT nMaxLength
);
```

The **GetContentType** method returns the content type for the current resource.

Parameters

hClient

Handle to the client session.

lpszContentType

A buffer which will contain the MIME content type of the resource returned by the web server. The buffer must be large enough to contain the entire content type string including the terminating null character. This parameter is optional and may be NULL if the information is not required.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer. If the *lpszContentType* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the value indicates the type of resource returned by the server. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Value	Description
HTTP_CONTENT_UNKNOWN	The content type for the resource could not be determined. This can occur if the content type returned by the server does not match a known value, such as an experimental or non-standard content type.
HTTP_CONTENT_BINARY	Application specific binary data which doesn't explicitly fall into any of the other standard MIME types. The most common type for binary data is application/octet-stream which can represent any type of data. Other examples are application/pdf and application/zip.
HTTP_CONTENT_TEXT	Textual data which can include plain text, HTML documents, XML, comma-separated value (CSV) data or any other human readable text formats. Examples are text/plain and text/html.
HTTP_CONTENT_AUDIO	Audio data which can include MEPG audio and Windows Media formats. Examples are audio/mpeg, audio/mp4 and audio/x-ms-wma.
HTTP_CONTENT_IMAGE	Image data which can include GIF, JPEG and PNG image formats. Examples are image/gif and image/jpeg.
HTTP_CONTENT_MESSAGE	A content type which specifies the response includes other messages, typically email messages in a standard MIME format. This content type is not commonly used with web services.

HTTP_CONTENT_MULTIPART	A content type which includes multiple types of data in a single response. The payload will typically be encoded text and the contents will need to be decoded to extract any binary data included in the message.
HTTP_CONTENT_VIDEO	Video data which can include MP4 movies or Window Media format video. Examples are video/mp4 and video/x-ms-wmv.
HTTP_CONTENT_FONT	Font data which can include TrueType and OpenType fonts. Examples are font/tff and font/otf.
HTTP_CONTENT_MODEL	Model data for 3D objects and scenes, such as those used with the Virtual Reality Modeling Language (VRML). Examples are model/3mf and model/vrml.

Remarks

This method does not examine the contents of the payload returned by the server and only returns a value based on the Content-Type response header. If the web server does not recognize the data format of the resource it is returning, it should identify it as **application/octet-stream** and the content type would be returned as HTTP_CONTENT_BINARY. However, some servers incorrectly return unrecognized formats as text/plain, causing the payload to be identified as human-readable text rather than binary data.

Some servers will return a content type of **text/plain** for JSON responses and others will use the IANA standard type of **application/json**. To ensure consistency with these two content types, this method will always return MIME_CONTENT_TEXT for JSON responses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetHeader, SetContentType, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetCookie Method

```
BOOL GetCookie(
   LPCTSTR lpszCookieName,
   LPTSTR lpszCookieValue,
   INT nCookieValue,
   LPTSTR lpszCookiePath,
   INT nCookiePath,
   LPTSTR lpszCookieDomain,
   INT nCookieDomain,
   LPSYSTEMTIME lpCookieExpires,
   LPDWORD lpdwCookieFlags
);

BOOL GetCookie(
   LPCTSTR lpszCookieName,
   CString& strCookieValue,
   CString& strCookiePath,
   CString& lpszCookieDomain,
   LPSYSTEMTIME lpCookieExpires,
   LPDWORD lpdwCookieFlags
);

BOOL GetCookie(
   LPCTSTR lpszCookieName,
   CString& strCookieValue
);
```

The **GetCookie** method returns information about a cookie set by the server.

Parameters

lpszCookieName

A null-terminated string which specifies the name of the cookie to return information about.

lpszCookieValue

A null-terminated string buffer that will contain the value of the cookie. If this information is not required, a NULL pointer may be specified.

nCookieValue

The maximum number of characters that may be copied into the buffer specified by the *lpszCookieValue* parameter, including the terminating null character.

lpszCookiePath

A null-terminated string buffer that will contain the cookie path. If this information is not required, a NULL pointer may be specified.

nCookiePath

The maximum number of characters that may be copied into the buffer specified by the *lpszCookiePath* parameter, including the terminating null character.

lpszCookieDomain

A null-terminated string buffer that will contain the cookie domain. If this information is not required, a NULL pointer may be specified.

nCookieDomain

The maximum number of characters that may be copied into the buffer specified by the

lpszCookieDomain parameter, including the terminating null character.

lpCookieExpires

A pointer to a SYSTEMTIME structure which specifies the date and time that the cookie expires. If this information is not required, a NULL pointer may be specified.

lpdwCookieFlags

One or more bit flags which specify status information about the cookie. A value of zero indicates that there are no special status flags for the cookie. This parameter may be NULL if the information is not required. The following values are currently defined:

Value	Description
HTTP_COOKIE_SECURE	This flag specifies that the cookie should only be provided to the server if the connection is secure.
HTTP_COOKIE_SESSION	This flag specifies that the cookie should only be used for the current application session and should not be stored permanently on the local system.

Return Value

If the method succeeds, the return value is non-zero. If the specified cookie does not exist or method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The Hypertext Transfer Protocol uses special tokens called "cookies" to maintain persistent state information between requests for a resource. These cookies are exchanged between the client and server by setting specific header fields. When a server wants the client to use a cookie, it will include a header field named Set-Cookie in the response header when the client requests a resource. The client can then take this cookie and store it, either temporarily in memory or permanently in a file on the local system. The next time that the client requests a resource from that server, it can send the cookie back to the server by setting the Cookie header field. The **GetCookie** method searches for a cookie set by the server in the Set-Cookie header field. The **SetCookie** method creates or modifies the Cookie header field for the next resource requested by the client.

There are two general types of cookies that are used by servers. Session cookies exist only for the duration of the client session; they are stored in memory and not saved in any kind of permanent storage. When the client application terminates, session cookies are deleted and no longer used. Persistent cookies are stored on the local system and are used by the client until their expiration time. It is the responsibility of the client application to store persistent cookies; applications may use a flat text file, a database or any other storage method available.

In addition to the cookie name and value, the server may return additional information about the cookie which the client should use to determine if it should send the cookie back to the server:

The ***cookie path*** specifies a path for the resources where the cookie should be used. For example, a path of "/" indicates that the cookie should be provided for all resources requested from the server. A path of "/data" would mean that the cookie should be included if the resource is found in the /data folder or a sub-folder, such as /data/projections.asp. However, the cookie would not be provided if the resource /info/status.asp was requested, since it is not in the /data path.

The ***cookie domain*** specifies the domain for which the cookie should be used. Matches are made by comparing the name of the server against the domain name specified in the cookie. If the domain is example.com, then any server in the example.com domain would match; for

example, both shipping.example.com and orders.example.com would match the domain value. However, if the cookie domain was orders.example.com, then the cookie would only be sent if the resource was requested from orders.example.com, not if the resource was located on shipping.example.com or www.example.com.

The ***cookie expiration*** specifies the date and time that the cookie should be deleted and no longer sent when a resource is requested from the server. This is only valid for persistent cookies, since session cookies are automatically deleted when the client application terminates. The time is always expressed as Coordinated Universal Time.

The ***cookie flags*** provide additional information about the cookie. In some cases, a cookie should only be submitted to the server if the resource is requested using a secure connection. In this case, the bit flag HTTP_COOKIE_SECURE will be set.

It is the responsibility of the client application to determine if a cookie meets the criteria required to be submitted to the server. If the application wishes to send the cookie, it can use the **SetCookie** method and specify the cookie name and value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

ClearHeaders, GetFirstCookie, GetNextCookie, GetHeader, SetCookie, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetData Method

```
INT GetData(
   LPCTSTR lpszResource,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);

INT GetData(
   LPCTSTR lpszResource,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);

INT GetData(
   LPCTSTR lpszResource,
   LPTSTR lpBuffer,
   DWORD dwMaxLength
);

INT GetData(
   LPCTSTR lpszResource,
   CString& strBuffer,
   DWORD dwOptions
);
```

The **GetData** method requests a resource from the server and copies the data to the specified buffer.

Parameters

lpszResource

A null-terminated string that specifies the resource that will be transferred to the local system. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the number of bytes of data returned by the server.

dwMaxLength

An unsigned integer value which specifies the maximum number of characters can be copied into the ***lpBuffer*** string. This parameter is used with the version of the method that returns the data in a character array and includes the terminating null character. This value must be greater than one and the ***lpBuffer*** parameter cannot be NULL, otherwise the method will return an error.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a

bitwise operator with any of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **EnableCompression** method.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetData** method is used to retrieve a resource from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpBuffer*** parameter point to a global memory handle which will contain the file data when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the third method or fourth method is used, where the data is returned in a string buffer, the data may be modified so that the end-of-line character sequence matches the convention used by the Windows platform (a carriage return character followed by a linefeed). If Unicode is being used, the data will be converted from a byte array to a Unicode string. An application should only use these versions of the **GetData** method if the resource or remote file contains text.

It is important for your application to initialize the value of the *lpdwLength* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the GET request to the server, the variable passed to this method may have a value of zero when the method returns, indicating no data has been returned to the caller.

When encountering a server error during a request, the **GetData** method normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. Additionally, servers may provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_TRANSFER_ERRORDATA option. When this option is enabled, the behavior of **GetData** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the buffer provided by the caller.

If compression has been enabled and the server returns compressed data, it will be automatically expanded before being returned to the caller. This will result in a difference between the value returned in the *lpdwLength* parameter, which contains the actual number of bytes copied into the buffer, and the values reported by **GetTransferStatus**. For example, if the server returns 5,000 bytes of compressed data that expands into 15,000 bytes, this function will return 15,000 as the number of bytes copied into the buffer. However, the **GetTransferStatus** method will return the content length as the original 5,000 bytes of compressed data. For this reason, you should always use the value returned in the *lpdwLength* parameter to determine the amount of data that has been copied into the buffer.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = pClient->GetData(lpszResource, &hgblBuffer, &cbBuffer);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
```

```
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetText, GetTransferStatus, PostData, PutData, PutFile, SubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetDefaultUserAgent Method

```
INT GetDefaultUserAgent(
   LPTSTR lpszUserAgent,
   INT nMaxLength
);

INT GetDefaultUserAgent(
   CString& strUserAgent
);
```

The **GetDefaultUserAgent** method returns the default user agent string which is included with all requests.

Parameters

lpszUserAgent

A buffer which will contain the default user agent string when the method returns. The buffer must be large enough to contain the entire user agent string including the terminating null character. This parameter cannot be NULL. An alternate form of the method accepts a **CString** variable which will contain the user agent.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When you submit a request to the server, a header named `User-Agent` is automatically defined which identifies your application. It typically uses a format where you have a product name and version number separated by a slash. There can be multiple products listed in the user agent string, with additional optional information enclosed in parenthesis.

A server uses the user agent to determine what type of client has issued the request, including what operating system and browser version was used. The **HttpGetDefaultUserAgent** method returns a copy of that default value. The value of the user agent string depends on which version of Windows the client is running on and whether the process is 32-bit or 64-bit. For example, a 64-bit application will typically return a value like this:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/134.0.0.0
Safari/537.36 SocketTools/12.0
```

Some web services check the value of the user agent string to determine if a compatible client is being used to issue the request. The default value is designed to emulate a common browser, but some services may require you change the user agent to use a specific value or include certain product names and versions.

You can change the value of the user agent string in one of two ways. You can use the **SetHeader** method to set the User-Agent header to a specific value for a client session, or you can use **SetDefaultUserAgent** to set the default user agent string for all requests. If you use **SetHeader** to change the user agent string for a client session, it will always override the default value.

To prevent any user agent string from being included with a request, include the HTTP_OPTION_NOUSERAGENT option when connecting to the server or use **SetOption** to enable that option after a connection has been established.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultUserAgent, SetDefaultUserAgent, SetOption, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetDefaultZoneId Method

```
BOOL GetDefaultZoneId(
   LPDWORD lpdwZoneId
);
```

The **GetDefaultZoneId** method returns the default security zone for downloaded files.

Parameters

lpdwZoneId

A pointer to an unsigned integer which is updated with the default zone ID for the client session. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the method succeeds, the return value is non-zero and *lpdwZoneId* will be updated with the default zone identifier. If no default zone has been explicitly set, the function will still return success and *lpdwZoneId* will be set to `SECURITY_ZONE_NONE`. If the method fails, it returns zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetDefaultZoneId** method returns the zone identifier which was previously set using the **SetDefaultZoneId** method. If no default zone has been specified for the client session, the default behavior is to not set a security zone for the file. This maintains backwards compatibility with previous versions of the library which would not create the security zone metadata for local files.

This method is typically used to confirm the currently configured zone before downloading a file, or to verify whether the default behavior has been overridden. The default zone ID is specific to the client session and does not affect other sessions or system-wide settings.

To get the security zone identifier for a local file, use the **GetFileZoneInfo** method. To change or remove the security zone information for a file, use the **SetFileZoneInfo** method.

For more detailed information about the zone identifiers, refer to the **SetDefaultZoneId** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetFileZoneInfo, SetDefaultZoneId, SetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetEncodingType Method

```
INT GetEncodingType();
```

The **GetEncodingType** method determines which content-encoding option is enabled.

Parameters

None.

Return Value

If the method succeeds, the return value is one of the following values:

Value	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request and no default content type will be specified. This encoding type should be used with REST APIs and other services which expect XML or JSON request payloads.
HTTP_ENCODING_URL	Non-printable and extended ASCII characters will be encoded so they can be safely used with URLs and form data. Encoded characters will be represented by a percent symbol prefix, followed by a two digit hexadecimal value which represents the ASCII character code. This encoding is typically used with web services which process HTML form data.
HTTP_ENCODING_XML	This encoding is identical to URL encoding, except spaces are not encoded. It is used with legacy web services which expect form data in an XML format and cannot process encoded whitespace. This encoding should not be specified for services which use REST APIs.

If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The default encoding type is HTTP_ENCODING_URL which is used for submitting data to services which expect HTML form data using the **HttpPostData** or **HttpSubmitForm** functions.

When submitting a JSON or XML request to a service using a REST API, your application should use HTTP_ENCODING_NONE and set the appropriate content type for the request payload. The HTTP_ENCODING_XML encoding type should only be used if the server expects URL encoded form data. The **PostJson** and **PostXml** methods will automatically set the correct encoding and content type for those requests.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetContentType, PostData, PostJson, PostXml, SetEncodingType, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFile Method

```
INT GetFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
);
```

The **GetFile** method downloads the specified file on the server to the local system.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **EnableCompression** method.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed. If this option is used, your application should call **HttpGetResultCode** to obtain the

	HTTP status code returned by the server. This will enable you to determine if the operation was successful.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. To determine if the server compressed the data returned to the client, use the **GetHeader** method to get the value of the **Content-Encoding** header after this function returns. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableCompression, EnableEvents, GetData, GetText, GetTransferStatus, PostData, PutData, PutFile, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFilePathUrl Method

```
INT GetFilePathUrl(
   LPCTSTR lpszFilePath,
   LPTSTR lpszFileUrl,
   INT nMaxLength
);

INT GetFilePathUrl(
   LPCTSTR lpszFilePath,
   CString& strFileUrl
);
```

The **GetFilePathUrl** method returns a normalized URL for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFilePath

A null-terminated string which specifies the file path on the server. The path may be absolute or relative, and the path does not have to exist on the server. Although this may be any type of resource, this method is primarily intended for use with static content such as documents and images. This parameter cannot be NULL.

lpszFileUrl

A null-terminated string buffer that will contain the URL for the specified file. If this parameter is NULL, nothing will be copied but the method will return the length of the URL, not including the terminating null character. An alternate version of this method accepts a reference to a **CString** object.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. This must include space for the terminating null character. If ***lpszFileUrl*** is NULL this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the ***lpszFileUrl*** buffer, not including the terminating null character. If a buffer is provided, it must be large enough to contain the entire URL or the method will fail. If the ***lpszFileUrl*** parameter is NULL, the method will return the length of the URL and nothing will be copied. If the method fails, it will return zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetFilePathUrl** method will return the URL associated with the specified file path. This can be useful if you wish to display or store URLs rather than relative paths to a file on the current server. For example, if you are connected to the server `example.com` on port 443 using a secure connection and you provide a file path such as `/documents/invoice.pdf` this method would return a URL which looks like `https://example.com/documents/invoice.pdf`. For security reasons, the resulting URL will never contain user credentials.

The file path in the URL will be normalized to remove extraneous directory traversals and path separators. It is not required for the file path to exist on the server at the time this method is called.

For more information about path normalization, refer to the **NormalizePath** method.

This method can also be used to obtain a URL which can be passed to the **GetSercurityZoneId** method to determine which security zone should be assigned to the file when it is downloaded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, GetSecurityZoneId, NormalizePath, NormalizeUrl, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFileSize Method

```
INT GetFileSize(
   LPCTSTR lpszFileName,
   LPDWORD lpFileSize
);

INT GetFileSize(
   LPCTSTR lpszFileName,
   ULARGE_INTEGER *lpFileSize
);
```

The **GetFileSize** method returns the size of the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpdileSize

Points to an unsigned integer that will contain the size of the specified file in bytes. For files larger than 4GiB, this parameter should point to an unsigned 64-bit integer.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. If the file does not exist or the client does not have permission to access the file, this method will fail and the ***lpFileSize*** parameter will be set to zero.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the method will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the ***lpdwFileSize*** parameter may still contain a valid value. If the library was able to determine the file size, but was not able to maintain the connection to the server, the returned file size will be greater than zero even if the method returns an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetContentLength, GetContentType, GetFileTime, VerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFileTime Method

```
INT GetFileTime(
   LPCTSTR lpszFileName,
   LPSYSTEMTIME lpFileTime,
   BOOL bLocalize
);
```

The **GetFileTime** method returns the modification time for the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a SYSTEMTIME structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetFileTime** method can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time.

This method uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a modification time for some resources. If the request to the server was successful and the file exists, but the server does not return a modification time, the method will succeed but all of the members of the SYSTEMTIME structure will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the modification time and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the SYSTEMTIME structure may still contain a valid value. If the library was able to determine the modification time, but was not able to maintain the connection to the server, the members of the SYSTEMTIME structure will specify a valid date and time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetFileSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFileZoneInfo Method

```
BOOL GetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   LPDWORD lpdwZoneId,
   LPTSTR lpszHostUrl,
   INT nMaxLength
);

BOOL GetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   LPDWORD lpdwZoneId,
   CString& strHostUrl
);
```

The **GetFileZoneInfo** method returns the security zone information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and read from the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

lpdwZoneId

A pointer to an unsigned integer which will contain the zone ID assigned to the file when the method returns. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string buffer which will contain the URL used to originally download the file. This information is not always available, in which case an empty string will be returned. If this information is not required, this parameter may be NULL and the ***nMaxLength*** parameter should be specified with a value of zero. An alternate version of this method accepts a reference to a **CString** object which will contain the URL when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the ***lpszHostUrl*** string including the terminating null character. If the ***lpszHostUrl*** parameter is NULL,

this value should be zero. If the buffer is not large enough to contain the entire URL, the string will be truncated. The maximum allowed URL length is 2,048 characters.

Return Value

If the method succeeds, the return value is non-zero. If the method fails or zone information is not available, the method returns zero. To get extended error information, call the **GetLastError** method.

Remarks

When a file is downloaded from the Internet or other external sources, Windows may assign a zone identifier to the file using an alternate data stream named `Zone.Identifier`. This metadata helps determine how the system treats the file with respect to security, such as whether to prompt the user with a warning before opening it.

The **GetFileZoneInfo** method retrieves the zone ID associated with the specified local file, allowing an application to determine the origin and trust level of that file. This can be useful when implementing additional security measures, auditing downloaded content, or providing user warnings about potentially untrusted files.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be read from files on network volumes because the SMB protocol does not support alternate data streams.

In addition to the zone ID, this method can optionally return the source URL used to download the file. This information, if present, is extracted from the same alternate data stream. Not all files will include this URL, as it depends on the program or browser that downloaded the file and whether it preserved this metadata. This method will not return an error if the URL stored with the security zone information is not a valid HTTP URL. For example, it's possible for this method to return zone information with a FTP URL. You can use the **ValidateUrl** method to determine if the host URL is a valid HTTP URL.

Note that not all file systems support alternate data streams. For example, FAT32 volumes do not support them, and in such cases, zone information will not be available. Additionally, some applications may remove this metadata after download or strip it during file copy operations. The absence of zone information does not guarantee that a file is safe or originated locally.

If you wish to distinguish between a file which exists without any zone information, and an error condition where the file does not exist or could not be accessed, check the last error code. If this method returns zero and the **GetLastError** method returns NO_ERROR, this indicates the file name is valid and it exists, but the `Zone.Identifier` alternate data stream does not exist.

For more detailed information about the security zone identifiers, refer to the **SetDefaultZoneId** method. To change or remove the security zone metadata from a local file, use the **SetFileZoneInfo** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetDefaultZoneId, SetDefaultZoneId, SetFileZoneInfo, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFirstCookie Method

```
BOOL GetFirstCookie(
   LPTSTR lpszCookieName,
   LPINT lpnNameLen,
   LPTSTR lpszCookieValue,
   LPINT lpnValueLen
);

BOOL GetFirstCookie(
   CString& strCookieName,
   CString& lpszCookieValue
);
```

The **GetFirstCookie** method returns the first cookie set by the server.

Parameters

lpszCookieName

A null-terminated string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A null-terminated string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds, the return value is non-zero. If there are no cookies or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstCookie** method is used to enumerate the cookies set by the server after a resource has been requested. To get information about a specific cookie, use the **GetCookie** method.

Example

```
CString strCookieName;
CString strCookieValue;
BOOL bResult;

// Get the first cookie set by the server
bResult = pClient->GetFirstCookie(strCookieName, strCookieValue);

while (bResult)
{
    // The strCookieName and strCookieValue strings contain the
```

```
    // the name and value for a cookie set by the server

    // Get the next cookie set by the server
    bResult = pClient->GetNextCookie(strCookieName, strCookieValue);
};
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetCookie, GetNextCookie, GetHeader, SetCookie, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFirstHeader Method

```
BOOL GetFirstHeader(
   LPTSTR lpszHeader,
   LPINT lpcchHeader
   LPTSTR lpszValue,
   LPINT lpcchValue
);

BOOL GetFirstHeader(
   CString& strHeader,
   CString& strValue
);
```

The **GetFirstHeader** method returns the name and value of the first response header.

Parameters

lpszHeader

A null-terminated string buffer that will contain the name of the header field when the method returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header name string.

lpszValue

A null-terminated string buffer that will contain the name of the header value when the method returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header value string.

Return Value

If the method succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

Use this method together with **GetNextHeader** to enumerate all request or response headers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetFormProperties Method

```
INT GetFormProperties(
   LPHTTPFORMPROPERTIES lpFormProp
);
```

The **GetFormProperties** function returns information about the current form.

Parameters

lpFormProp

Points to a HTTPFORMPROPERTIES structure which will contain information about the current form.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateForm, SetFormProperties, SubmitForm, HTTPFORMPROPERTIES

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetHeader Method

```
BOOL GetHeader(
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxLength
);

BOOL GetHeader(
   LPCTSTR lpszHeader,
   CString& strValue
);
```

The **GetHeader** method returns the value of the specified response header field.

Parameters

lpszHeader

A pointer to a null-terminated string which specifies the name of the HTTP header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszValue

Pointer to a buffer which will contain the null-terminated string value of the specified header field. This may also be a **CString** object which will contain the value of the header field when the function returns.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

When a resource is returned by the server, it consists of three parts. The first part consists of a single line that indicates the result of the request. The second part is one or more header fields which provides specific information about the resource, such as its size in bytes. The third part consists of the resource data itself, such as the HTML document or image data. For example, this is what the response to a request for a simple HTML document can look like:

```
HTTP/1.0 200 OK
Date: Mon, 5 Jan 2004 20:18:33 GMT
Content-Type: text/html
Last-Modified: Mon, 5 Jan 2004 19:34:19 GMT
Content-Length: 115

<html>
<head>
<title>Simple Document</title>
</head>
<body>
This is a simple HTML document.
```

```
<body>
</html>
```

The first line consists of the protocol version, a numeric response code and some text describing the result. The subsequent lines are the header, which is similar to the headers used in email messages. For example, the Date field specifies the date the resource was requested, the Content-Type field specifies what type of resource was requested, and the Content-Length field specifies the size of the resource in bytes. The end of the header block is indicated by an empty line (two carriage-return/linefeed sequences), and is followed by the resource itself, in this case a simple HTML document.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCookie, GetFirstCookie, GetFirstHeader, GetNextCookie, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetNextCookie

```
DWORD GetNextCookie(
   LPTSTR lpszCookieName,
   LPINT lpnNameLen,
   LPTSTR lpszCookieValue,
   LPINT lpnValueLen
);

DWORD GetNextCookie(
   CString& strCookieName,
   CString& strCookieValue
);
```

The **GetNextCookie** method returns the next cookie set by the server.

Parameters

lpszCookieName

A null-terminated string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A null-terminated string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds, the return value is non-zero. If there are no more cookies or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextCookie** method is used to enumerate the cookies set by the server after a resource has been requested. To get information about a specific cookie, use the **GetCookie** method.

Example

```
CString strCookieName;
CString strCookieValue;
BOOL bResult;

// Get the first cookie set by the server
bResult = pClient->GetFirstCookie(strCookieName, strCookieValue);

while (bResult)
{
    // The strCookieName and strCookieValue strings contain the
```

```
    // the name and value for a cookie set by the server

    // Get the next cookie set by the server
    bResult = pClient->GetNextCookie(strCookieName, strCookieValue);
};
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetCookie, GetFirstCookie, GetHeader, SetCookie, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetNextHeader Method

```
BOOL GetNextHeader(
   LPTSTR lpszHeader,
   LPINT lpcchHeader,
   LPTSTR lpszValue,
   LPINT lpcchValue
);

BOOL GetNextHeader(
   CString& strHeader,
   CString& strValue
);
```

The **GetNextHeader** method returns the name and value of the next response header.

Parameters

lpszHeader

A null-terminated string buffer that will contain the name of the header field when the method returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header name string.

lpszValue

A null-terminated string buffer that will contain the name of the header value when the method returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header value string.

Return Value

If the method succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

Use this method iteratively after **GetFirstHeader** to enumerate all request or response headers.

Unlike the **GetHeader** method, which returns a single header name and value, the **GetFirstHeader** and **GetNextHeader** methods will return multiple headers that have the same common name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, GetFirstHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetOption Method

```
INT GetOption(
   DWORD dwOption,
   LPBOOL lpbEnabled
);
```

The **GetOption** method determines whether a specified HTTP option is enabled.

Parameters

dwOption

An unsigned integer which specifies the option that is to be checked. It may be one of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is

	established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpbEnabled

A pointer to an integer which will be set to a non-zero value when the method returns if the specified option has been enabled. If the option has not been enabled, a zero value will be returned in the variable.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**. Note that the value returned in the ***lpbEnabled*** parameter is only valid if the method succeeds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Connect, SetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetPriority Method

```
INT GetPriority();
```

The **GetPriority** method returns a value which specifies the priority of file transfers.

Parameters

None.

Return Value

If the method succeeds, the return value is the current file transfer priority. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Value	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetRedirectUrl Method

```
INT GetRedirectUrl(
   LPTSTR lpszResource,
   INT nMaxLength
);

INT GetRedirectUrl(
   CString& strResource
);
```

The **GetRedirectUrl** method returns the URL for a redirected resource.

Parameters

lpszResource

A null-terminated string buffer which will contain the URL of the redirected resource when the method returns. The string will be null terminated and the buffer must be large enough to accommodate the entire URL or the method will fail. This parameter cannot be a NULL pointer. An alternate form of the method accepts a **CString** variable which will contain the URL.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the length of the URL. A return value of zero indicates that the request was not redirected. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns the complete URL for a request which has been redirected to a different resource. This is useful for applications which do not wish to use automatic redirection and instead prefer to handle the redirection by establishing a new connection by passing this value to the **ConnectUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, GetHeader GetResourceUrl,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetResourceUrl Method

```
INT GetResourceUrl(
   LPTSTR lpszResource,
   INT nMaxLength
);

INT GetResourceUrl(
   CString& strResource
);
```

The **GetResourceUrl** method returns the URL for the current request.

Parameters

lpszResource

A null-terminated string buffer which will contain the URL of the current resource when the method returns. The string will be null terminated and the buffer must be large enough to accommodate the entire URL or the method will fail. This parameter cannot be a NULL pointer. An alternate form of the method accepts a **CString** variable which will contain the URL.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the length of the URL. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns the URL for the current request which has been submitted to the server. This can be useful for applications which need to compose a URL for display or logging purposes, combining the connection information and resource path provided to methods such as **GetData** or **PostData**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, GetData, GetRedirectUrl, PostData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Command, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetResultString Method

```
INT GetResultString(
   LPTSTR lpszResult,
   INT cbResult
);

INT GetResultString(
   CString& strResult
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

The buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetSecurityZoneId Method

```
BOOL GetSecurityZoneId(
   LPCTSTR lpszResource,
   LPDWORD lpdwZoneId
);
```

The **GetSecurityZoneId** method returns the security zone identifier for the specified resource.

Parameters

lpszResource

A null-terminated string which specifies the resource URL to check. This must be either a valid HTTP URL or a local file URL.

lpdwZoneId

A pointer to an unsigned integer which is updated with the zone ID for the specified resource. It may be one of the following values when the method returns:

Value	Description
SECURITY_ZONE_NONE	The security zone for this resource cannot be determined.
SECURITY_ZONE_LOCAL	The resource is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The resource originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The resource originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The resource originates from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The resource originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the method succeeds, the return value is non-zero and *lpdwZoneId* will be updated with the zone identifier. If the security zone could not be determined, *lpdwZoneId* will be set to `SECURITY_ZONE_NONE`. If the method fails, it returns zero and the error code can be retrieved by calling the **GetLastError** method.

Remarks

This **GetSecurityZoneId** method is used to determine how a resource should be handled in terms of local security policy before performing a file transfer or download. For example, you may want to determine whether a resource originates from the Internet or local intranet before deciding whether to attach zone metadata, prompt the user, or apply additional security restrictions. It can also be used to implement policy-based rules for accepting or rejecting downloads based on their origin.

The *lpszResource* parameter must use the `http://`, `https://` or `file://` schemes. This method will not accept URLs for other schemes such as `ftp://` or `sftp://`. While this method does not verify that the resource exists, it does require the URL to be syntatically valid. If the host name cannot be resolved to an IPv4 or IPv6 address, the method will fail. If the resource is a local file, it must be a regular file

and cannot be the name of a folder or device.

If the resource is specified using a `file://` URL that refers to a file on a network share (e.g.: `file://server/share/file.txt`), the method will attempt to access that file across the network. This may cause the application to block while Windows resolves the hostname and establishes a connection to the shared volume. If the network share is unavailable or slow to respond, the method may block until the connection succeeds or the attempt times out.

If this method returns FALSE, this indicates either the resource type is not supported or there was an internal failure when the Internet Security Manager attempted to map the resource URL to a security zone. In most default configurations, this method will return the zone `SECURITY_ZONE_INTERNET` for a valid URL unless the host has been explicitly added to the **Intranet Zone** or **Trusted Sites** list on the local system.

Security zones are used with Mark of the Web (MOTW) metadata which indicates if a file originated from an external source, such as the Internet or an untrusted network. When a file is downloaded using a web browser or other applications, it may add a `Zone.Identifier` alternate data stream to the file. This stream contains information such as `ZoneId=3`, which corresponds to the Internet Zone. Your application can use the **GetFileZoneInfo** method to retrieve this information, which allows you to determine whether a file was downloaded from an external source and what zone Windows has associated with it. This metadata affects how files are handled by Windows security components such as Microsoft Defender, SmartScreen, Office Protected View, and script execution policies in Windows Explorer. If the file has no metadata, it is generally treated as coming from the Local Machine Zone (zone 0). If your application sets or clears this metadata using the **SetFileZoneInfo** method, it can influence how other applications interpret the file's trust level.

It is important to note that this method is used to determine what security zone would be appropriate for a given resource, and it only performs syntactic checks and classification of the URL using the **IInternetSecurityManager** COM interface. The security zone this method returns is based on the current system and user policy, using zone assignments stored in the registry. Typically this information is managed using Windows settings or Internet options in the control panel. This method does not read the zone identifier metadata which has been assigned to a file.

For more detailed information about the zone identifiers, refer to the **SetDefaultZoneId** method.

Example

```
LPCTSTR pszDownloadUrl = _T("https://example.com/pub/download/file.txt");
DWORD dwZoneId = SECURITY_ZONE_NONE;

if (httpClient.GetSecurityZoneId(pszDownloadUrl, &dwZoneId))
{
    // dwZoneId now contains SECURITY_ZONE_INTERNET
    _tprintf(_T("The security zone is: %u\n"), dwZoneId);
}
else
{
    DWORD dwError = httpClient.GetLastError();
    // Handle the error
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshttpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultZoneId, GetFileZoneInfo, SetDefaultZoneId, SetFileZoneInfo, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
HTTP_STATUS_UNUSED	No connection has been established.
HTTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
HTTP_STATUS_CONNECT	The client is establishing a connection with the server.
HTTP_STATUS_READ	The client is reading data from the server.
HTTP_STATUS_WRITE	The client is writing data to the server.
HTTP_STATUS_DISCONNECT	The client is disconnecting from the server.
HTTP_STATUS_GETDATA	The client is downloading data from the server.
HTTP_STATUS_PUTDATA	The client is uploading data to the server.
HTTP_STATUS_POSTDATA	The client is posting data to a script on the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

IsBlocking, IsInitialized, IsReadable, IsWritable, GetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetText Method

```
INT GetText(
   LPCTSTR lpszResource,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **GetText** method copies the contents of a text file on the server, or the text output from a script, to the specified string buffer.

Parameters

lpszResource

A null-terminated string that specifies the resource that will be transferred to the local system. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A null-terminated string buffer which will contain the contents of the text file when the method returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetText** method is used to download a text file or retrieve the text output from a script and store the contents in a string buffer. Because binary data can include embedded null characters which would truncate the string, this method should only be used with text files or script output that is known to be textual. For example, it is safe to use this method when a resource returns HTML or XML data, but should not be used if it returns an image or executable file.

This method has been included as a convenience for applications that need to retrieve relatively small amounts of textual data and manipulate the contents as a string. If the Unicode version of this method is called, the text is automatically converted to a Unicode string. If the maximum amount of data being returned is unknown or the amount of text is very large, it is recommended that you use the **GetData** or **GetFile** methods.

If you use the **GetFileSize** method to determine how large the string buffer should be prior to calling this method, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **GetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line

characters.

HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = pHttpClient->GetText(lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshttpv10.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetData, GetFile, GetTransferStatus, PutData, PutFile, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both the requests submitted and the responses returned by the server. The initial timeout value is set when establishing a connection and the default timeout period is 20 seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetTransferStatus Method

```
INT GetTransferStatus(
   LPHTTPTRANSFERSTATUS lpStatus
);

INT GetTransferStatus(
   LPHTTPTRANSFERSTATUSEX lpStatus
);
```

The **GetTransferStatus** method returns information about the current data transfer in progress.

Parameters

lpStatus

A pointer to an HTTPTRANSFERSTATUS or HTTPTRANSFERSTATUSEX structure which contains information about the status of the current data transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this method will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

EnableEvents, GetStatus, RegisterEvent, HTTPTRANSFERSTATUS, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::NormalizePath Method

```
INT NormalizePath(
   LPCTSTR lpszResource,
   LPTSTR lpszNormalized,
   INT nMaxLength
);

INT NormalizePath(
   LPCTSTR lpszResource,
   CString& strNormalized
);
```

The **NormalizePath** method normalizes a resource on the server returning a canonical path.

Parameters

lpszResource

A null-terminated string which specifies a resource on the server. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized resource path when the method returns. The buffer must be large enough to contain the entire path including the terminating null character. If this parameter is NULL it will be ignored and the method will return the length of the normalized path without copying any characters. An alternate version of this method accepts a reference to a **CString** object which will contain the normalized path when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the method will return the length of the normalized path without copying any characters.

Return Value

This method returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized path but will not copy any characters. If the method fails, it will return HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will accept a resource path on the server and return a normalized version of that path. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. For example, if the resource path provided is:

```
/text/..//project//./files/filename.pdf
```

This method would return the following as the normalized path:

```
/project/files/filename.pdf
```

The extraneous path delimiters are removed (the double slashes) and the reference to the parent directory is collapsed to produce a simple, uniform path. This can be useful both for display purposes and to normalize input provided by a user. This method will accept an HTTP URL and return only the normalized resource path for the URL. If you need to normalize a complete URL, use the **NormalizeUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NormalizeUrl, ValidateHostName ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::NormalizeUrl Method

```
INT NormalizeUrl(
   LPCTSTR lpszUrl,
   LPTSTR lpszNormalized,
   INT nMaxLength
);

INT NormalizeUrl(
   LPCTSTR lpszUrl,
   CString& strNormalized
);
```

The **NormalizeUrl** method normalizes the resource path and returns a canonical URL.

Parameters

lpszUrl

A null-terminated string which specifies an HTTP URL. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the method returns. The buffer must be large enough to contain the entire URL including the terminating null character. If this parameter is NULL it will be ignored and the method will return the length of the normalized URL without copying any characters. An alternate version of this method accepts a reference to a **CString** object which will contain the normalized path when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the method will return the length of the normalized URL without copying any characters.

Return Value

This method returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized URL but will not copy any characters. If the method fails, it will return HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will accept a standard HTTP or HTTPS URL and return a canonical version of that URL. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. If non-standard port numbers are included in the URL, they will be preserved. For example, if the follwing URL is provided to the method:

```
https://www.example.com:443/file/..//project/filename.pdf
```

It will return the following as the normalized URL:

```
https://www.example.com/project/filename.pdf
```

Because the port number specified in the original URL is the default port number for secure HTTP connections, it is removed. In addition, the path is collapsed and the extraneous path delimiters are removed.

This method is primarily intended to normalize URLs which may be entered by a user or read from a configuration file. It is not necessary to call this method prior to calling **ConnectUrl** because it will

always be normalized internally prior to the connection being established. If you wish to check a URL to ensure it is valid and the server name can be resolved to an IP address, use the **ValidateUrl** method.

If you need to normalize a remote file path, use the **NormalizePath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, NormalizePath, ValidateHostName ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::OpenFile Method

```
INT OpenFile(
   LPCTSTR lpszFileName
);
```

The **OpenFile** method opens the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to open.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Only one file may be opened at a time for each client session. Use the **CloseFile** method to close the file on the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PatchData Method

```
INT PatchData(
   LPCTSTR lpszResource,
   LPCTSTR lpszPatchData,
   LPBYTE lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PatchData(
   LPCTSTR lpszResource,
   LPCTSTR lpszPatchData,
   HGLOBAL *lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PatchData(
   LPCTSTR lpszResource,
   LPCTSTR lpszPatchData,
   CString& strResult,
   DWORD dwOptions
);
```

The **PatchData** method submits patch data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name that the patch data will be submitted to. Typically this is the name of a script on the server.

lpszPatchData

A null-terminated string that specifies the patch data that will be submitted to the server.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. An alternate version of this method accepts a **CString** object which will contain the server response.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description

HTTP_PATCH_DEFAULT	The default patch mode. The contents of the buffer will be submitted without encoding. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_PATCH_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_PATCH_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PatchData** method is used to submit XML or JSON formatted patch data to a service, and then returns a copy of the response from the server into a local buffer. This method will not perform any encoding and will not automatically define the type of patch data being submitted. Your application is responsible for specifying the content type for the patch data, and ensuring that the XML or JSON data that is being submitted to the server is formatted correctly.

This method sends a PATCH command to the server, which is similar to a POST or PUT request. It is used to make partial updates to a resource, rather than creating or replacing it entirely. The format of the patch data is specific to the service being used. If the resource being patched does not exist, the behavior is defined by the server. If enough information is provided, it may choose to create the resource just as if a PUT command was used, or it may return an error.

Your application should use the **SetHeader** method to define the Content-Type header prior to calling the **PatchData** method. One of the most common formats used is the JSON Merge Patch which is defined in RFC 7396. The value for the Content-Type header for this patch format is "application/merge-patch+json". Refer to your service API documentation to determine what patch formats are acceptable, along with any additional header values that must be defined.

The method may be used in one of two ways, depending on the needs of the application. The first approach is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the ***lpvResult*** parameter will point to the buffer that was allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second approach is to have the ***lpvResult*** parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpcbResult*** parameter correctly. This

parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the PATCH request to the server, the variable passed to this method may have a value of zero when the method returns, indicating no data has been returned to the caller.

When encountering a server error during a request, the **PatchData** method normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the PATCH request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_PATCH_ERRORDATA option. When this option is enabled, the behavior of **PatchData** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of the transaction while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_PATCH_CONVERT option is used

nResult = lpClient->PatchData(lpszResource,
                              lpszPatchData,
                              &hgblBuffer,
                              &cbBuffer,
                              HTTP_PATCH_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetData, GetTransferStatus, PostData, PostJson, PostXml, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PostData Method

```
INT PostData(
   LPCTSTR lpszResource,
   LPBYTE lpBuffer,
   DWORD cbBuffer,
   LPBYTE lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostData(
   LPCTSTR lpszResource,
   LPBYTE lpBuffer,
   DWORD cbBuffer,
   HGLOBAL *lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostData(
   LPCTSTR lpszResource,
   LPCTSTR lpszBuffer,
   CString& strResult,
   DWORD dwOptions
);
```

The **PostData** method submits the contents of the specified buffer to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A null-terminated string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script. This string may specify a valid URL for the current server that the client is connected to.

lpBuffer

The data that will be provided to the script. This parameter may be NULL if the script does not require any additional data from the client. In an alternate form of the method, this may point to a string which contains the data to be posted.

cbBuffer

The number of bytes to copy from the buffer. If the ***lpBuffer*** parameter is NULL, this value should be zero.

lpResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. An alternate version of this method accepts a **CString** object which will contain the server response.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_MULTIPART	The contents of the buffer being sent to the server consists of multipart form data and will be sent as-is without any encoding.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostData** method is used to submit data to a script that executes on the server and then copy the output from that script into a local buffer. If you are submitting XML formatted data to the server, it is recommended that you use the **PostXml** method to ensure that the correct content type and encoding is automatically selected.

The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the ***lpvResult*** parameter will point to the buffer that was allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvResult*** parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpcbResult*** parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this method may have a value of zero when the method returns, indicating no

data has been returned to the caller.

When encountering a server error during a request, the **PostData** method normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **PostData** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = lpClient->PostData(lpszResource,
                             lpParameters,
                             cbParameters,
                             &hgblBuffer,
                             &cbBuffer,
                             HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetData, GetTransferStatus, PostJson, PostXml, PutData, SubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PostFile Method

```
INT PostFile(
   LPCTSTR lpszFileName,
   LPCTSTR lpszResource,
   LPCTSTR lpszFieldName
);
```

The **PostFile** method posts the contents of the specified file to a script executed on the server.

Parameters

lpszFileName

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszResource

A null-terminated string that specifies the resource on the server that the file data will be posted to. Typically this is the name of a script that is responsible for processing and storing the file data.

lpszFieldName

A null-terminated string that corresponds to the form field name that the script expects. If this parameter is NULL or an empty string, a default field name of "File1" is used.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostFile** method is similar to the **PutFile** method in that it can be used to upload the contents of a local file to a server. However, instead of using the PUT command, the POST command is used to send the file data to a script that is executed on the server. This method has the advantage of not requiring any special configuration settings on the server, however it does require that the script be able to process multipart/form-data as defined in RFC 2388.

To support uploading files from a form on a webpage, the FILE input type is used along with the action that specifies the script that will accept the file data and process it. For example, the HTML code could look like this:

```
<form action="/cgi-bin/upload.cgi" method="post" enctype="multipart/form-
data">
<input type="file" name="datafile" size="20">
<input type="submit">
</form>
```

In this example, the script /cgi-bin/upload.cgi is responsible for processing the file data that is posted by the client, and the form field name "datafile" is used. The user can select a file, and when the Submit button is clicked, the file data is posted to the script. To simulate this using the **PostFile** method, the *lpszFileName* parameter should be set to the name of the local file that will be posted to the server. The *lpszResource* parameter should be the name of the script, in this case "/cgi-bin/upload.cgi". The *lpszFieldName* parameter should be specified as the string "datafile" to match the name of the field used by the form.

Note that the **PostFile** method always submits the file contents as multipart/form-data with the content type set to application/octet-stream. The script that accepts the posted data must be able to

parse the multipart header block and correctly process 8-bit data. If the script assumes that the data will be posted using a specific encoding type such as base64 then the file data may not be accepted or may be corrupted by the script.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetData, GetFile, GetTransferStatus, PostData, PutData, PutFile, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PostJson Method

```
INT PostJson(
   LPCTSTR lpszResource,
   LPCTSTR lpszJsonData,
   LPBYTE lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostJson(
   LPCTSTR lpszResource,
   LPCTSTR lpszJsonData,
   HGLOBAL *lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostJson(
   LPCTSTR lpszResource,
   LPCTSTR lpszJsonData,
   CString& strResult,
   DWORD dwOptions
);
```

The **PostJson** method submits JSON structured data to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A null-terminated string that specifies the resource name the JSON data will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszJsonData

A null-terminated string which specifies the JSON structured text to be submitted to the server. If the Unicode version of this method is called, the text will sent as a UTF-8 encoded string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the number of bytes of data returned by the server.

strResult

An alternate version of this method accepts a reference to a **CString** object which will contain the response from the server when the method returns. If the method fails, an empty string will be returned. This version of the method should only be used when you are certain the response from the server will not contain binary data. If the Unicode version of this method is called, it will automatically convert a UTF-8 encoded response to a UTF-16 Unicode string.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostJson** method is used to submit JSON structured data to a REST API endpoint or script and then returns the output in a local buffer. This method automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure it is well-formed. Your application is responsible for ensuring the JSON data being submitted to the server is formatted correctly.

The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the ***lpvResult*** parameter will point to the buffer that was allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvResult*** parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpcbResult*** parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this method may have a value of zero when the method returns, indicating no data has been returned to the caller.

If the content type for the request has not been explicitly specified, it will be automatically updated by this method to use the "application/json" content type. You can override the default content type for the request by calling the **SetHeader** method prior to calling this method. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

When encountering a server error during a request, the **PostJson** method normally returns

HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **PostJson** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the JSON payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the server to respond.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc method; the handle to this memory will be
// returned in the hgblBuffer parameter.

nResult = lpClient->PostJson(lpszResource,
                             lpszJsonData,
                             &hgblBuffer,
                             &cbBuffer,
                             HTTP_POST_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, GetResultCode, GetText, PostText, PostXml, PutText, SetHeader, SubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PostText Method

```
INT PostText(
   LPCTSTR lpszResource,
   LPCTSTR lpszTextData,
   LPBYTE lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostText(
   LPCTSTR lpszResource,
   LPCTSTR lpszTextData,
   HGLOBAL *lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostText(
   LPCTSTR lpszResource,
   LPCTSTR lpszTextData,
   LPTSTR lpszResult,
   LPDWORD lpcchResult,
   UINT nCodePage,
   DWORD dwOptions
);

INT PostText(
   LPCTSTR lpszResource,
   LPCTSTR lpszTextData,
   CString& strResult,
   UINT nCodePage,
   DWORD dwOptions
);
```

The **PostText** method submits unstructured text data to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A null-terminated string that specifies the resource name the text data will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszTextData

A null-terminated string which specifies the text to be submitted to the server. If the Unicode version of this function is called, the text will sent as a UTF-8 encoded string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the method

returns, this value will be updated with the number of bytes of data returned by the server.

strResult

An alternate version of this method accepts a reference to a **CString** object which will contain the response from the server when the method returns. If the method fails, an empty string will be returned. This version of the method should only be used when you are certain the response from the server will not contain binary data. If the Unicode version of this method is called, it will automatically convert a UTF-8 encoded response to a UTF-16 Unicode string.

nCodePage

An optional unsigned integer that specifies the ANSI code page which should be used to convert the text returned by the server. A value of zero specifies the default system code page should be used with the conversion. In most cases, applications should use CP_UTF8 as the code page. See the remarks below for more information.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostText** method is used to submit plain text to a REST API endpoint or script and then returns the output in a local buffer. This method can be used to submit any kind of text payload to the server, and will automatically sets a default content type without any encoding, ensuring the text is submitted exactly as specified. If the Unicode version of this method is called, the text will be converted to UTF-8 prior to being submitted to the server. The endpoint or script on the server side must be capable of processing UTF-8 encoded text correctly or you may experience unexpected results.

If the content type for the request has not been explicitly specified, it will be automatically updated by this method to use the generic "text/plain" content type. You can override the default content type for the request by calling the **SetHeader** method prior to calling this method. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type. Your application should ensure any content type you specify is valid for the contents of the payload; the content type is not checked against the text you submit to

the server.

It is important for your application to initialize the value of the *lpcbResult* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this method may have a value of zero when the method returns, indicating no data has been returned to the caller.

If you use a version of this method which return the response in a string buffer, you should typically specify the *nCodePage* parameter as **CP_UTF8** because this is what most servers use as their default character set. If the server is returning text using a character set other than UTF-8, specifying a code page enables your application to automatically normalize that text to Unicode. It is important to note the text returned by this function will always be either UTF-16 or UTF-8 Unicode to ensure there is no data loss during the conversion process.

When encountering a server error during a request, the **PostText** method normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **PostText** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the text payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the server to respond.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc method; the handle to this memory will be
// returned in the hgblBuffer parameter.

nResult = lpClient->PostText(lpszResource,
                             lpszTextData,
                             &hgblBuffer,
                             &cbBuffer,
                             HTTP_POST_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
```

```
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, GetResultCode, GetText, PostJson, PostXml, PutText, SetHeader, SubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PostXml Method

```
INT PostXml(
   LPCTSTR lpszResource,
   LPCTSTR lpszXmlData,
   LPBYTE lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostXml(
   LPCTSTR lpszResource,
   LPCTSTR lpszXmlData,
   HGLOBAL *lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT PostXml(
   LPCTSTR lpszResource,
   LPCTSTR lpszXmlData,
   CString& strResult,
   DWORD dwOptions
);
```

The **PostXml** method submits XML structured data to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A null-terminated string that specifies the resource name the XML data will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszXmlData

A null-terminated string which specifies the JSON structured text to be submitted to the server. If the Unicode version of this method is called, the text will sent as a UTF-8 encoded string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

strResult

An alternate version of this method accepts a reference to a **CString** object which will contain the response from the server when the method returns. If the method fails, an empty string will be returned. This version of the method should only be used when you are certain the response from the server will not contain binary data. If the Unicode version of this method is called, it will automatically convert a UTF-8 encoded response to a UTF-16 Unicode string.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostXml** method is used to submit XML structured data to a REST API endpoint or script and then returns the output in a local buffer. This function automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure it is well-formed. Your application is responsible for ensuring the XML data being submitted to the server is formatted correctly.

The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the ***lpvResult*** parameter will point to the buffer that was allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvResult*** parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpcbResult*** parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this method may have a value of zero when the method returns, indicating no data has been returned to the caller.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "text/xml" content type. You can override the default content type for the request by calling the **SetHeader** method prior to calling this method. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

When encountering a server error during a request, the **PostXml** method normally returns

HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **PostXml** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the XML payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the server to respond.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter.

nResult = lpClient->PostXml(lpszResource,
                            lpszXmlData,
                            &hgblBuffer,
                            &cbBuffer,
                            HTTP_POST_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, GetResultCode, GetText, PostJson, PostText, PutText, SetHeader, SubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ProxyConnect Method

```
BOOL ProxyConnect(
   UINT nProxyType,
   LPCTSTR lpszProxyHost,
   UINT nProxyPort,
   LPCTSTR lpszProxyUser,
   LPCTSTR lpszProxyPassword,
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion
);

BOOL ProxyConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion
);
```

The **ProxyConnect** method is used to establish a connection through a proxy server.

Parameters

nProxyType

An unsigned integer which specifies the type of proxy that the client is connecting to. The supported proxy server types are as follows:

Value	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the *dwOptions* parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server settings configured in Windows.

lpszProxyHost

A null-terminated string which specifies the proxy server host name or IP address. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy

configuration has been specified for the local system.

nProxyPort

The port number that the proxy server is listening for connections on. A value of zero specifies that the default port number 80 should be used. Note that in most cases, a proxy server is not configured to use the default port. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

lpszProxyUser

A null-terminated string which specifies the user name that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszProxyPassword

A null-terminated string which specifies the password that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszRemoteHost

A null-terminated string which specifies the name of the server to connect to through the proxy server. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.

HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use

	of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create a version value and the following values are defined:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.
HTTP_VERSION_09	The client should use the original one-line protocol which includes no version number and no request header block. This version has been deprecated and should only be used with legacy servers which do not support the protocol standard. This version of the protocol does not support virtual hosts and is not supported by most modern web services.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.
HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2.0 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with

	header compression and optimizing how requests are serviced.
HTTP_VERSION_30	The client should use the HTTP/3 protocol standard defined in RFC 9114. This protocol version is similar to earlier versions of the protocol in terms the request methods and server responses, but the underlying transport mechanism is significantly different. Instead of using TCP, this version of the protocol uses multiplexed UDP connections designed to reduce the overhead of TCP while maintaining reliability. If the client or server does not support HTTP/3, the client will automatically attempt to use an earlier version of the protocol.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **ProxyConnect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

If the HTTP_PROXY_WINDOWS proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related parameters will be ignored and the function will establish a connection directly to the server. The second form of the **ProxyConnect** method always uses the system configuration to determine if the connection should be made through a proxy server, which is why there are no parameters such as the proxy type or proxy server name.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2 or HTTP/3.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Applications should use HTTP_VERSION_DEFAULT as the value for the *dwVersion* parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2

or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return zero.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

Example

```
CHttpClient *pClient = new CHttpClient();

// Establish a connection using the default proxy server
// configuration for the local system

if (pClient->ProxyConnect(strHostName) == FALSE)
{
    pClient->ShowError();
    return;
}

// Retrieve the resource from the server and store it
// in the string buffer

nResult = pClient->GetData(strResource, strBuffer);

if (nResult == HTTP_ERROR)
    pClient->ShowError();

pClient->Disconnect();
delete pClient;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, ConnectUrl, Disconnect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PutData Method

```
INT PutData(
   LPCTSTR lpszRemoteFile,
   LPBYTE lpBuffer,
   DWORD dwLength
);

INT PutData(
   LPCTSTR lpszRemoteFile,
   LPCTSTR lpszBuffer
);
```

The **PutData** method copies the contents of the specified buffer and stores it in a file on the server.

Parameters

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created. This string may specify a valid URL for the current server that the client is connected to.

lpBuffer

A buffer which will contain the data to be transferred from the client and stored in a file on the server. In an alternate form of the method, this may be a null terminated string.

dwLength

The number of bytes to copy from the buffer.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutData** method is used to store the contents of the specified buffer in a file on the server. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **Authenticate** method prior to uploading the data.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetData, GetFile, GetTransferStatus, PostData, PutFile, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PutFile Method

```
INT PutFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile
);
```

The **PutFile** method transfers the specified file on the local system to the server.

Parameters

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutFile** method is used to transfer a file from the local system to a server. Not all servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **Authenticate** method prior to uploading the file.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetData, GetFile, GetTransferStatus, PostData, PostFile, PutData, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::PutText Method

```
INT PutText(
   LPCTSTR lpszRemoteFile,
   LPCTSTR lpszBuffer
);
```

The **PutText** method creates a text file on the server using the contents of a string buffer.

Parameters

lpszRemoteFile

A null-terminated string that specifies the text file on the server that will be created or overwritten. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A null-terminated string that contains the text that will be stored in the file.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutText** method is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **Authenticate** method prior to uploading the data.

If the Unicode version of this method is called, the string will be converted to ASCII and then uploaded to the server. If you wish to store the contents of the string as Unicode on the server, you must use the **PutData** method. This method should never be used to upload binary data.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshttpv10.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetText, GetTransferStatus, PutData, PutFile, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
HTTP_EVENT_CONNECT	The connection to the server has completed.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **HttpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a resource is being retrieved or data is being submitted to the server. The HTTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one HTTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The ***dwParam*** parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, GetTransferStatus HttpEventProc,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetBearerToken Method

```
INT SetBearerToken(
   LPTSTR lpszBearerToken
);
```

The **SetBearerToken** method sets the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

lpszBearerToken

A null-terminated string buffer which contains the bearer token used to authorize client requests. If this parameter is NULL or a zero-length string, the current bearer token will be cleared and no client authentication will be performed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Using OAuth 2.0 requires you to understand the process of how to request the bearer (access) token. Obtaining a bearer token requires registering your application with the web service provider, getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access the service.

Your application should not store a bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Authenticate, GetBearerToken, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetContentType Method

```
BOOL SetContentType(
   LPCTSTR lpszContentType
);
```

The **SetContentType** method sets the content type for the next request.

Parameters

hClient

Handle to the client session.

lpszContentType

A null-terminated string which contains the MIME content type of the data being submitted to the web server. If this parameter is NULL or an empty string, the current type will be cleared and the client will use a default type based on the request and content encoding.

Return Value

If the method succeeds, it will return a non-zero value. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

The content type string must use the standard IANA format which consists of a type and subtype separated by a forward slash. Some common content types are:

Content Type	Description
`text/plain`	The request payload is unstructured text in a human readable format. This content type serves a default for textual data which does not explicitly have another standard subtype.
`text/csv`	The request payload is in a comma-separated value (CSV) text format.
`application/octet-stream`	The request payload contains binary data. This content type serves a default for binary data which does not explicitly have another standard subtype.
`application/json`	The request payload is standard JSON formatted text. This is the default content type used when submitting JSON requests, however some services may expect the client to use the non-standard `text/x-json` type.
`application/xml`	The request payload is standard XML formatted text. This is the default content type used when submitting XML requests, however some servers may expect the client use the deprecated `text/xml` type. The standard content type for XML is defined in RFC 7303, however most services will recognize both types as valid.

If a content type is not explicitly specified with a request, an appropriate default content type will be

automatically selected. For example, the **PostJson** method will set the content type to **application/json** by default. This method can be used to override that default value.

This method will validate the content type, but does not validate the subtype. Unregistered subtypes should always be prefixed with "x-" which designates it as experimental. Some commonly used types also use this designation, such as **text/x-vcalendar** for the vCalendar format.

If your application needs to set the content type to a non-standard value for a proprietary API, use the **SetHeader** method to set the Content-Type header value. That method will not perform any validation on the header value. The contents of the payload will not be checked to ensure the content type is valid. It is the responsibility of the application to ensure the correct content type is used when submitting the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetContentType, GetHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetCookie Method

```
BOOL SetCookie(
   LPCTSTR lpszCookieName,
   LPCTSTR lpszCookieValue
);
```

The **SetCookie** method sends the specified cookie to the server when a resource is requested.

Parameters

lpszCookieName

Pointer to a string which specifies the name of the cookie that will be sent to the server when the next resource is requested.

lpszCookieValue

Pointer to a string which specifies the value of the cookie. To delete a cookie that has been previously set, this parameter should be NULL or point to an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetCookie** method submits the cookie name and value to the server when a resource is requested or data is posted to a script. For more information about cookies and how they are used, refer to the **GetCookie** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

ClearHeaders, GetCookie, GetFirstCookie, GetNextCookie, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetDefaultUserAgent Method

```
INT SetDefaultUserAgent(
   LPCTSTR lpszUserAgent
);
```

The **SetDefaultUserAgent** method sets the default user agent string which is included with all requests.

Parameters

lpszUserAgent

A null-terminated string which specifies the new default user agent string. If this parameter is NULL or an empty string, the client will revert to using the default user agent string defined when the library was first initialized.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **SetDefaultUserAgent** method changes the default user agent string which is included with each request made by the client. This default value is global and changing it will change the default user agent for all client sessions in all threads created by the process.

The user agent string should use a format with the product name and version number separated by a slash. There can be multiple products listed in the user agent string separated by spaces, with additional optional information enclosed in parenthesis. For example, a valid user agent string for an application named "MyProgram" could look like this:

```
MyProgram/1.0 (Win32)
```

Some web services check the value of the user agent string to determine if a compatible client is being used to issue the request. The default value used by SocketTools is designed to emulate a common browser, but some services may require you change the user agent to use a specific value or include certain product names and versions.

To change the user agent string for a specific client session, use the **SetHeader** method to set a value for the `User-Agent` header. To get the current value of the default user agent string, use the **GetDefaultUserAgent** method.

To prevent any user agent string from being included with a request, include the HTTP_OPTION_NOUSERAGENT option when connecting to the server or use **SetOption** to enable that option after a connection has been established.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultUserAgent, SetOption, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetDefaultZoneId Method

```
BOOL SetDefaultZoneId(
   DWORD dwZoneId
);
```

The **SetDefaultZoneId** method sets the default security zone for downloaded files.

Parameters

dwZoneId

An unsigned integer which specifies the default security zone identifier. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it. This typically indicates that the file was created locally or has not been marked by Windows as originating from another zone. This is the default behavior if the application does not explicitly set a default security zone.
SECURITY_ZONE_LOCAL	The file is considered to be part of the **Local Machine** zone, which includes files stored on the user's local system or removable media. This zone has the least restrictive security settings, and files are typically allowed to run without prompting or limitation. ***This zone should not be assigned to downloaded files***, as doing so can bypass security mechanisms designed to protect against potentially harmful content. Only use this setting for files that are created locally by the application itself and are known to be safe. Misuse of this setting can result in a significant security risk, especially if the file contains scripts, executables, or other active content.
SECURITY_ZONE_INTRANET	The file originates from the **Local Intranet** zone, which includes content served from internal network resources, such as shared drives, file servers, and internal web applications. This zone typically has fewer security restrictions than the Internet zone but more than the Local Machine zone. It is commonly used in enterprise environments where network resources are managed and considered relatively safe. ***This zone should not be used for files downloaded from the public Internet***, as doing so may incorrectly signal that the file is trusted within the organization. Assign this zone only to files that are retrieved from authenticated internal sources or known secure locations on the local network.
SECURITY_ZONE_TRUSTED	The file is from a site or source explicitly added to the **Trusted Sites** zone in Internet Explorer or Windows security settings. This zone is intended for domains or locations that the user or system administrator considers

	to be safe, such as internal applications hosted on the public Internet or third-party services with a strong reputation for safety. Files marked with this zone typically bypass certain security checks, and applications may allow active content (such as scripts or macros) to run without prompting. Use caution when assigning this zone to downloaded files, as it may override protections that would otherwise apply to untrusted content.
SECURITY_ZONE_INTERNET	The file was downloaded from the **Internet** zone, which represents untrusted or unknown sources outside the local network. This is the standard zone identifier automatically applied by web browsers, email clients, and many applications when a file is downloaded from a public website or external server. Windows uses this zone to enforce security restrictions, such as prompting the user before executing scripts, blocking active content, or displaying a warning when the file is opened. This is typically the appropriate zone to assign for any file obtained over HTTP from an external site. Assigning this zone helps ensure that users are notified before opening potentially unsafe content, in accordance with Windows Attachment Manager and Group Policy settings.
SECURITY_ZONE_RESTRICTED	The file is associated with the **Restricted Sites** zone, which enforces the strictest security policies. This zone is used for content that is explicitly marked as untrusted or potentially harmful. Files tagged with this zone may be blocked from execution, prevent active content from running, or trigger warnings when accessed. Assigning this zone can be useful for sandboxing unverified downloads or limiting the risk from files obtained from questionable sources. However, it may also cause legitimate files to fail to open or operate with reduced functionality depending on system policy and application behavior.

Return Value

If the method succeeds, the return value is non-zero. If the method fails it will return zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SetDefaultZoneId** method sets the default security zone identifier that will be applied to all files downloaded during the client session. This setting controls how Windows treats downloaded files with respect to security prompts and content execution restrictions. Once the default zone ID is set, it will be automatically applied to all subsequent file downloads performed by the client session. This eliminates the need to manually set the security zone for each individual file, ensuring consistent zone tagging across the session.

When `SECURITY_ZONE_NONE` is specified, no `Zone.Identifier` alternate data stream is created. This causes the file to be treated as if it originated from the local system, and Windows will not apply zone-based security restrictions. This is the default behavior which maintains backwards compatibility with previous versions of the library that did not create the security zone metadata for files.

The zone ID metadata is interpreted by Windows components such as **Attachment Execution Services** and **SmartScreen** to determine whether the user should be warned when opening a file. In enterprise environments, Group Policy settings may alter how different zone IDs are handled, such as blocking scripts, macros, or unsigned executables. Developers should test zone assignment behavior in target environments when distributing files to end users.

Note that this method does not retroactively modify existing files. The default zone identifier is applied only at the time a file is downloaded using methods such as **DownloadFile**.

If your application is distributed in a controlled environment, such as a corporate network with endpoint protection or user access controls, consult your organization's security policy before modifying default zone behavior. Improper zone tagging can impact file access, generate security prompts, or interfere with digital signature enforcement.

Be aware that some file operations (such as extracting from a ZIP archive, or copying files to non-NTFS volumes or across certain network shares) may strip the `Zone.Identifier` metadata. Developers should not rely solely on zone tagging as a persistent trust mechanism outside of the initial download context.

To get the security zone identifier for a local file, use the **GetFileZoneInfo** method. To change or remove the security zone information for a file, use the **SetFileZoneInfo** method.

Assigning `SECURITY_ZONE_LOCAL` as the default security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

DownloadFile, GetDefaultZoneId, GetFileZoneInfo, SetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetEncodingType Method

```
INT SetEncodingType(
   INT nEncodingType
);
```

The **SetEncodingType** method specifies the type of encoding to be applied to the content of a HTTP request.

Parameters

nEncodingType

Specifies the content encoding type for the request. The following values are defined:

Value	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request and no default content type will be specified.
HTTP_ENCODING_URL	Non-printable and extended ASCII characters will be encoded so they can be safely used with URLs and form data. Encoded characters will be represented by a percent symbol prefix, followed by a two digit hexadecimal value which represents the ASCII character code. This encoding is typically used with web services which process HTML form data. If the content type has not been explicitly specified, it will default to **application/x-www-form-urlencoded**.
HTTP_ENCODING_XML	This encoding specifies the content being submitted to the server contains XML structured data. If the content type has not been defined, it will default to **application/xml**, which is the official MIME media type for XML text. Legacy services may expect **text/xml** as the content type.
HTTP_ENCODING_JSON	This encoding specifies the content being submitted to the server contains JSON structured data. If the content type has not been defined, it will default to **application/json**, which is the official MIME media type for JSON text. In rare cases, legacy services may expect **text/x-json** as the content type.
HTTP_ENCODING_UTF8	This encoding specifies the content is unstructured text which has been UTF-8 encoded. If the content type has not been defined, it will default to **text/plain; charset=utf-8** which is appropriate for any kind of general text payloads. If you use the Unicode version of the methods in this library, all text is normally submitted as UTF-8 encoded text.
HTTP_ENCODING_ANSI	This encoding specifies the content is unstructured text and may not be UTF-8 encoded if the ANSI version of the methods in this library are used. This encoding is not recommended unless there is a specific need to send text using a specific character set. If the content type has not been defined, it will default to **plain/text**.

HTTP_ENCODING_XMLURL	This encoding is identical to URL encoding, except spaces are not encoded. It is used with legacy web services which expect form data in an XML format and cannot process encoded whitespace. This encoding should not be specified for services which use REST APIs. If the content type has not been specified, it will default to `application/x-www-form-urlencoded`.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetEncodingType** method sets the type of encoding used when optional parameter data is submitted with a request for a resource. If you do not set the Content-Type request header prior to submitting the request, the default content type provided to the server will be determined by the encoding you choose. If you specify HTTP_ENCODING_NONE, no default content type will be provided. Because most services require a Content-Type header as part of a POST request, this can cause the server to reject the request. The **SetContentType** method can be used to explicitly specify the content type for the request.

If the service you are connected with expects an XML formatted request, it is recommended you use the **PostXml** method. If the service expects JSON formatted text, use the **PostJson** method. These methods ensure the correct encoding and content type are always used. If the service expects some other kind of text payload, you can set the appropriate content type using the **SetContentType** method and then use the **PostText** method to submit the request.

If you are using the **PostData** method to submit form data to the server, in most cases you should use HTTP_ENCODING_URL. If the form data contains XML text, you may need to use the HTTP_ENCODING_XMLURL encoding type. However, in most cases, this is not the appropriate encoding type for XML and should never be used when submitting XML to a REST API endpoint.

If you are submitting a payload which contains binary data, you would typically use HTTP_ENCODING_NONE and explicitly set the appropriate content type using the **SetContentType** method. For example, if you're submitting a JPEG image to a REST endpoint, you should not use any encoding and set the Content-Type header to `image/jpeg`. If your application needs to set the content type to a non-standard value for a proprietary API, use the **SetHeader** method to set the Content-Type header value. That method will not perform any validation on the header value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetEncodingType, PostData, PostJson, PostText, PostXml, SetContentType, SetHeader, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetFileZoneInfo Method

```
BOOL SetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   DWORD dwZoneId,
   LPCTSTR lpszHostUrl
);
```

The **SetFileZoneInfo** method sets or clears the zone identifier information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and write to the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

dwZoneId

An unsigned integer which specifies the zone ID which should be assigned to the file. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string which specifies the URL used to originally download the file. If this information is not required the parameter may be a NULL pointer or an empty string. If an originating URL is provided, it must be a valid HTTP URL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails it will return zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SetFileZoneInfo** method enables applications to set or remove the security zone identifier metadata associated with a local file. This information is stored in an alternate data stream named `Zone.Identifier` and is used by Windows to determine the origin and trust level of a file, potentially prompting the user with a warning before the file is opened.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This

includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be applied to files on network volumes because the SMB protocol does not support alternate data streams. If a file with zone metadata is copied from a local NTFS drive to a network share, any associated streams like `Zone.Identifier` will be lost.

Specifying `SECURITY_ZONE_NONE` will remove any existing zone identifier stream from the file. This has the same effect as selecting the Unblock option in the file's properties dialog in Windows Explorer. Once the stream is removed, the file will be treated as though it originated from the local computer. Note that when the `Zone.Identifier` stream is deleted from a file to remove its security zone information, Windows Explorer may not immediately reflect this change. This is because Explorer caches certain file metadata, including zone information used to display the "This file came from another computer" warning in the file's Properties dialog.

When assigning a zone ID, it is important to ensure that the value accurately reflects the source of the file. For files that have been downloaded from the Internet, it is strongly recommended to use `SECURITY_ZONE_INTERNET`. This allows Windows and other software to enforce appropriate security measures, such as requiring user confirmation before executing or opening the file.

In most scenarios, zone identifiers should only be set for files that were downloaded from untrusted or external sources. Setting an overly permissive zone (such as `SECURITY_ZONE_LOCAL`) for files obtained from the Internet or other potentially unsafe locations can expose users to security risks and circumvent built-in protections.

If a URL is provided via the *lpszHostUrl* parameter, it will be included in the zone metadata and can be retrieved later using the **GetFileZoneInfo** method. This provides additional context about the origin of the file and may aid in auditing or diagnostic tasks. The URL must be a syntactically valid HTTP address and should reflect the actual location from which the file was downloaded.

Not all file systems support alternate data streams. The method will fail if used on a volume (such as FAT32) that does not support this feature. Additionally, modifying zone information may fail if the file is in use, read-only, or if the current process does not have the required permissions. It is possible to disable the creation of security zone metadata through local policy settings. In this case, the method will fail and the last error code will be set to `ST_ERROR_POLICY_NO_ZONEINFO`.

Assigning `SECURITY_ZONE_LOCAL` as the security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetDefaultZoneId, GetFileZoneInfo, SetDefaultZoneId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetFormProperties Method

```
INT SetFormProperties(
   LPHTTPFORMPROPERTIES lpFormProp
);
```

The **SetFormProperties** function updates the properties of the current form.

Parameters

lpFormProp

Points to a HTTPFORMPROPERTIES structure which specifies the new properties for the current form.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateForm, GetFormProperties, SubmitForm, HTTPFORMPROPERTIES

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetHeader Method

```
BOOL SetHeader(
   LPCTSTR lpszHeader,
   LPCTSTR lpszValue
);
```

The **SetHeader** method sets the value of the specified request header field.

Parameters

lpszHeader

A pointer to a null-terminated string which specifies the name of the HTTP header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszValue

A pointer to a null-terminated string which specifies the value of the HTTP header. Headers values cannot contain carriage-return or linefeed characters. If the value contains Unicode characters, they will be UTF-8 encoded. If this parameter or NULL or a zero-length string, the header field is deleted from the request headers.

Return Value

If the method succeeds, the return value is non-zero. If the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

The **SetHeader** method defines a header field and value that is submitted to the server when a resource is requested. If the header field has already been defined, it will be replaced by the new value. There are a number of header fields which are automatically created by the library, and others that are conditionally created depending on whether or not certain options are specified. The following header fields are automatically created by the library:

Header Field	Description
Accept	This header specifies the types of resources that are acceptable to the client. By default, all resource data types are accepted by the client.
Authorization	This header is automatically created if the client uses the Authenticate method to authenticate a client session.
Connection	This header determines if the connection is maintained after a resource has been requested, or if the connection should be immediately closed. The value of this header depends on whether the HTTP_OPTION_KEEPALIVE option has been specified.
Content-Length	This header defines the length of the data that is being posted to the server or the size of a file being uploaded to the server.
Content-Type	This header defines the content type for data posted to the server. This header is created if URL encoding is specified. The specific type of encoding used can be set by calling the SetEncodingType method.

Host	This header specifies the name of the server that the client has connected to. This is automatically generated when any resource is requested.
Pragma	This header is used to control caching performed by the server. If the option HTTP_OPTION_NOCACHE has been specified, this header will automatically be defined with the value "no-cache".
Proxy-Authorization	This header is automatically created if a proxy connection has been established and a username and password is required to authenticate the client session.

Request headers are generated by methods that send resource requests. In some cases, header values are supplied by the requesting method only if the application has not previously defined the header. For others, the requesting method overrides what the application may have defined.

If you use this method to set the Authorization header to a custom value for this client session, you must not call the **Authenticate** method. The **Authenticate** method will always override any custom Authorization header value and replace it with the credentials token generated from the username and password provided.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Authenticate, GetFirstHeader, GetHeader, GetNextHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetOption Method

```
INT SetOption(
   DWORD dwOption,
   BOOL bEnabled
);
```

The **SetOption** method enables or disables the specified option.

Parameters

hClient

Handle to the client session.

dwOption

An unsigned integer which specifies one of the following options:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function

	may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

bEnabled

An integer value which determines if the option should be enabled or disabled. A non-zero value specifies that the option should be enabled, while a zero value specifies that the option should be disabled.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetOption** method can only enable or disable the options listed above. Other options are only available when the connection is initially established. For example, you cannot use this function to enable security options or switch to using HTTP/2 after the connection has been made.

If you use HTTP_OPTION_FREETHREAD to permit any thread to reference a client handle allocated in another thread, your application is responsible for ensuring it does not attempt to submit requests using the same handle on different threads at the same time. If two different threads attempt to perform an operation using the same handle, there is no guarantee as to which thread will complete the operation first.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Connect, GetOption, ProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetPriority Method

```
INT SetPriority(
   INT nPriority
);
```

The **SetPriority** method specifies the priority for file transfers.

Parameters

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Value	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the method succeeds, the return value is the previous file transfer priority. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetPriority** method can be used to control the processor usage, memory and network bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the

process, the HTTP_PRIORITY_BACKGROUND value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The HTTP_PRIORITY_CRITICAL priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application.

If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to HTTP_PRIORITY_CRITICAL, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SetTimeout Method

```
INT SetTimeout(
   INT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete. If this value is zero, the default timeout period of 20 seconds will be used.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both the requests submitted and the responses returned by the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SubmitForm Method

```
INT SubmitForm(
   LPBYTE lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT SubmitForm(
   HGLOBAL *lpResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);

INT SubmitForm(
   CString& strResult,
   DWORD dwOptions
);
```

The **SubmitForm** method submits the contents of the current form to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the lpvBuffer parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_SUBMIT_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_SUBMIT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value

is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SubmitForm** method is used to submit form data to a script that executes on the server and then copy the output from that script into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** function.

To determine the current status of a transaction while it is in progress, use the **GetTransferStatus** function.

Example

```
CString strResult;
INT nResult = 0;

pClient->CreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);
pClient->AddField(_T("UserName"), lpszUserName);
pClient->AddField(_T("Password"), lpszPassword);

nResult = pClient->SubmitForm(strResult);
pClient->DestroyForm();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddField, AddFile, ClearForm, CreateForm, DeleteField, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::SubmitRequest Method

```
INT SubmitRequest(
   DWORD dwRequestType,
   LPCTSTR lpszResource,
   LPHTTPCONNECTION lpConnection,
   LPCVOID lpvRequest,
   DWORD dwRequestLength,
   LPVOID lpvResponse,
   LPDWORD lpdwResponseLength
);
```

The **SubmitRequest** method provides a high-level interface for submitting requests to a web service.

Parameters

dwRequestType

An unsigned integer which specifies the type of request being submitted to the server. It may be one of the following values:

Value	Description
HTTP_METHOD_DEFAULT	Perform a default request action based on the value of parameters passed to this method. If there is no request payload, the action will default to HTTP_METHOD_GET, otherwise it will default to HTTP_METHOD_POST. It is generally recommended that you explicitly specify the request method; however, this default value can be useful if your application does not know if a payload will be included with the request at the time the method is called.
HTTP_METHOD_GET	Use the GET method when submitting the request to the server. If this action is used, the ***lpvRequest*** and ***dwRequestLength*** parameters will be ignored. The buffer specified by the ***lpvResponse*** parameter will contain the server's response to the request.
HTTP_METHOD_POST	Use the POST method when submitting the request to the server. The payload specified by the ***lpvRequest*** parameter will be included with the request and the buffer specified by the ***lpvResponse*** parameter will contain the server's response to the request.
HTTP_METHOD_PUT	Use the PUT method when submitting the request to the server. The payload specified by the ***lpvRequest*** parameter will be included with the request. This method is typically used with persistent storage on the server, such as uploading the contents of a file. If the ***lpvResponse*** parameter is not NULL, it will contain any response from the server after the operation has completed.

lpszResource

A null-terminated string which specifies the complete URL for the resource. Both standard and secure connections are supported, and query parameters may be included with the URL. This

parameter cannot be NULL and must specify an absolute (fully qualified) URI which begins with either http:// or https://.

lpConnection

A pointer to a **HTTPCONNECTION** structure which contains additional information about the client connection. This structure is required if your application must connect through a proxy server or when authentication is required to access a resource. This parameter can be NULL, in which case default values will be used when establishing the connection.

lpvRequest

A buffer which contains the payload data submitted to the server as part of the request. This parameter may point to a byte buffer or a null-terminated string. If there is no payload required for the request this parameter may be NULL. If the Unicode version of this method is called and the request buffer contains text, it will be automatically converted to UTF-8 prior to submitting the data to the server. If the ***dwRequestLength*** parameter is -1, the payload will always be processed as a null-terminated string and the method will fail if the buffer does not contain text characters.

dwRequestLength

An unsigned integer which specifies the size of the request payload. If the payload contains text, this should specify the number of characters in the buffer. If the payload contains binary data, this value should specify the number of bytes in the buffer. If the ***lpvRequest*** parameter points to a null-terminated string, this value can be -1 and the length of the payload will be determined automatically. If the Unicode version of this method is called and the length is -1, the request payload will be UTF-8 encoded prior to being submitted to the server.

lpvResponse

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. This parameter may be NULL if the response data is not required. If this parameter is NULL, the ***lpdwResponseLength*** parameter must also be NULL. If a response buffer is provided, the data will always be returned as a byte stream, even when the Unicode version of this method is called. If the server returns a text payload, it will typically be UTF-8 encoded and the application can convert the text to Unicode using the **MultiByteToWideChar** method.

lpdwResponseLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResponse*** parameter. If the ***lpvResponse*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of data that was returned. If the ***lpvResponse*** parameter is not NULL, this parameter cannot be NULL.

Return Value

If the method succeeds, the return value is the server status code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call the **GetLastError** method. The application should always check the return value to determine if the request was successful. If the server returns a 4xx or 5xx status code, this indicates the request was not accepted and the response buffer will typically contain information about the cause of the failure.

Remarks

The **SubmitRequest** method provides a single high-level interface for submitting requests to a web server and returning the server's response. This method is entirely self-contained and does not require a client session handle. The current thread will block until the method returns and will not generate event notifications. If you need to download a large file from a web server and want progress updates,

it is recommended you use the **DownloadFile** method.

If the service you are using requires authentication, you will need to allocate a **HTTPCONNECTION** structure and set the appropriate member values to specify the authentication method and user credentials. This structure also allows you to provide additional information, such as additional request header values, a custom timeout period and additional connection options.

When the *lpvRequest* parameter is not NULL, the contents of the buffer will be examined to determine if it contains text or binary data. If the *dwRequestLength* parameter is -1, the *lpvRequest* parameter is always considered to be a pointer a null-terminated string, with the length calculated by counting the number of characters up to the terminating null character. If the request payload is text, the method will also attempt to determine if the payload looks like XML or JSON and will automatically set the appropriate content type. The application can explicitly provide a MIME content type by setting the *lpszContentType* member of the **HTTPCONNECTION** structure.

Example

```
// Submit an XML payload to the server and return the response in a global
// memory buffer allocated by the method
LPCTSTR lpszResource = _T("https://www.example.com/postxml");
LPCTSTR lpszRequest = _T("<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n") \
                      _T("<test>\r\n") \
                      _T("  <data param=\"A\">Value1</data>\r\n") \
                      _T("  <data param=\"B\">Value2</data>\r\n") \
                      _T("</test>\r\n");

// Provide additional connection information which includes user credentials
// and a custom header which includes an API token for the service; note that
// we need to initialize the dwSize member of the structure
HTTPCONNECTION httpConnection = { 0, };
httpConnection.dwSize = sizeof(httpConnection);
httpConnection.nAuthType = HTTP_AUTH_BASIC;
httpConnection.lpszUserName = _T("userid");
httpConnection.lpszPassword = _T("secret");
httpConnection.lpszContentType = _T("application/xml");
httpConnection.lpszHeaders = _T("X-API-Token: 99d2fe39-0246-4efa-98e0-
4d775579fa5d");

// Declare the global memory handle which will contain the server response
// and initialize the length to 0, which tells the API that a HGLOBAL handle
// is being used rather than a pointer to a pre-allocated buffer
CHttpClient httpClient;
HGLOBAL hgblResponse = NULL;
DWORD dwLength = 0;

INT nResult = httpClient.SubmitRequest(
    HTTP_METHOD_POST,
    lpszResource,
    &httpConnection,
    lpszRequest,
    (DWORD)-1,
    &hgblResponse,
    &dwLength);

if (nResult != HTTP_ERROR && hgblResponse != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResponse);
```

```
    // The lpBuffer variable points to the data returned by the server and
    // the application is responsible for freeing the global memory handle

    GlobalUnlock(hgblResponse);
    GlobalFree(hgblResponse);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetData, PostData, PostJson, PostXml, UploadFile, HTTPCONNECTION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::UploadFile Method

```
BOOL UploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPHTTPTRANSFERSTATUS lpStatus
   HTTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **UploadFile** method uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.

HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **UploadFile** method provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then upload the file to the server. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, then function will attempt to change the current working directory. Note that the path in an HTTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

Not all web servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the URL specify a username and password to upload a file.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the method will fail and return a value of zero.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer

as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **UploadFile** method is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **PutFile** to upload the file. If the server does not support the PUT command, you may be able to upload files using the **PostFile** method. Refer to that method for more information.

Example

```
CHttpClient httpClient;
CString strLocalFile = _T("c:\\temp\\database.mdb");
CString strFileURL =
_T("http://update:secret@www.example.com/updates/database.mdb");

if (!httpClient.UploadFile(strLocalFile, strFileURL))
{
    httpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetFileZoneInfo, GetTransferStatus, HttpEventProc, PostFile, PutFile, SetFileZoneInfo, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ValidateHostName Method

```
BOOL ValidateHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszAddress,
   INT nMaxLength
);

BOOL ValidateHostName(
   LPCTSTR lpszHostName,
   CString& strAddress
);
```

The **ValidateHostName** method determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The method will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the ***nMaxLength*** parameter is zero, it will be ignored and the IP address will not be returned. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the ***lpszAddress*** string buffer. The buffer must be large enough to store the complete address. Because this method can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the ***lpszAddress*** parameter will be ignored.

Return Value

If the method succeeds, the host name is valid and the return value will be non-zero. If the method fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ValidateHostName** method provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. If the Unicode version of this method is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

If the ***lpszHostName*** parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address.

If you wish to validate a complete HTTP URL instead of a host name, use the **ValidateUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, ValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::ValidateUrl Method

```
BOOL CHttpClient::ValidateUrl(
    LPCTSTR lpszUrl
);
```

The **ValidateUrl** method determines if a string represents a valid HTTP URL.

Parameters

lpszUrl

A null-terminated string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **ValidateUrl** method will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure HTTP scheme. This method will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this method will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **ConnectUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, DownloadFile, UploadFile, ValidateHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::VerifyFile Method

```
BOOL VerifyFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile
);
```

The **VerifyFile** method attempts to verify that the size of a file on the local system is the same as the specified file on the server.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of file on the local system.

lpszRemoteFile

A null-terminated string that specifies the name of the file on the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **VerifyFile** method will attempt to verify that the contents of the local and remote files are identical by comparing the size of the files. This method is provided for compatibility with the File Transfer Protocol API, and should not be considered a reliable method for comparing files. Web servers may not consistently return file size information for dynamically created content such as HTML pages which use server-side includes.

It is not recommended that you use this method with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **VerifyFile** method to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteFile, GetFile, GetFileSize, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a null-terminated string is passed as the buffer argument. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Data Structures

- HTTPCONNECTION
- HTTPFORMPROPERTIES
- HTTPTRANSFERSTATUS
- HTTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPFORMPROPERTIES

This structure is used by the **GetFormProperties** and **SetFormProperties** methods to return and modify the properties of the specified form.

```
typedef struct _HTTPFORMPROPERTIES
{
   UINT    nFormMethod;
   UINT    nFormType;
   LPCTSTR lpszFormAction;
   DWORD   dwReserved1;
   DWORD   dwReserved2;
} HTTPFORMPROPERTIES, *LPHTTPFORMPROPERTIES;
```

Members

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. It may be one of the following values:

Value	Description
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. It may be one of the following values:

Value	Description
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

lpszFormAction

A null-terminated string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

dwReserved1

A reserved structure member.

dwReserved2

A reserved structure member.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateForm, GetFormProperties, SetFormProperties, SubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPTRANSFERSTATUS

This structure is used by the **GetTransferStatus** method to return information about a data transfer in progress.

```
typedef struct _HTTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} HTTPTRANSFERSTATUS, *LPHTTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

EnableEvents, GetTransferStatus, RegisterEvent, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPTRANSFERSTATUSEX

This structure is used by the **GetTransferStatus** method to return information about a data transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _HTTPTRANSFERSTATUSEX
{
   ULARGE_INTEGER uiBytesTotal;
   ULARGE_INTEGER uiBytesCopied;
   DWORD          dwBytesPerSecond;
   DWORD          dwTimeElapsed;
   DWORD          dwTimeEstimated;
} HTTPTRANSFERSTATUSEX, *LPHTTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

EnableEvents, GetTransferStatus, RegisterEvent, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Queue Class Library

Manage a queue which will perform HTTP file transfers in the background.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

File Name	CSHTPV12.DLL
Version	12.0.3024.1286
LibID	E5D90316-2B5F-4E36-BD24-F58F1B47C3BF
Import Library	CSHTPV12.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 7230, RFC 7540

Overview

The Hypertext Transfer Queue class library provides a high-level interface to uploading and downloading a collection of files using the Hypertext Transfer Protocol. Queued file transfers are performed in a background worker thread, with optional event notifications to inform the application about the current state of the queue and the files being processed.

A file may be queued for upload or download, and both uploads and downloads may be included in the same queue. Multiple instances of this class may be created to transfer multiple files at the same time. This class supports the same options available to individual file transfers using the CHttpClient class.

This class supports HTTP versions 1.0, 1.1, 2.0 and 3.0. Secure connections use TLS 1.2 or later and supports the use of client certificates if required. There are several methods designed to make it easy to submit JSON and XML requests and include custom headers which may be required by REST APIs. The class also supports standard authentication methods and the use of OAuth bearer tokens.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Queue Class Methods

Class	Description
CHttpQueue	Constructor which initializes the current instance of the class
~CHttpQueue	Destructor which releases resources allocated by the class
Method	Description
AddFile	Add a file to the transfer queue
AttachHandle	Attach the specified client handle to this instance of the class
Cancel	Cancel the specified queued file transfer
Clear	Remove all files from the transfer queue
DetachHandle	Detach the handle for the current instance of this class
DisableTrace	Disable logging of network function calls
EnableTrace	Enable logging of network function calls to a file
EnumFiles	Enumerate queued files based on the specified criteria
FindFile	Find a specific file in the queue
HttpEventProc	Callback function that processes events generated during queued transfers
GetClient	Return a handle to the active client session during a queued file transfer
GetCount	Return the number of files queued for transfer
GetErrorString	Return a description for the specified error code
GetFile	Return information about the specified file
GetFirstFile	Return information about the first file in the transfer queue
GetHandle	Return a handle for the queue
GetLastError	Return the last error code
GetNextFile	Return information about the next file in the transfer queue
GetStatus	Return the current status of the file transfer queue
GetTransferStatus	Return information about the file being currently transferred
IsEmpty	Determine if the current file transfer queue is empty
IsIdle	Determine if the queue manager is currently idle
IsInitialized	Determine if the class has been successfully initialized
RemoveFile	Remove a file from the transfer queue
Reset	Reset the internal state of file transfers in the queue
Resume	Resume the transfer of files after queue processing has been paused
SetLastError	Set the last error code
ShowError	Display a message box with a description of the specified error
Start	Begin transferring files in the queue

Stop	Stop transferring files in the queue
Suspend	Pause the transfer of files in the queue
Wait	Wait for the transfer of all queued files to complete

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::CHttpQueue Method

```
CHttpQueue();

CHttpQueue(
   UINT nMaxFiles,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **CHttpQueue** constructor initializes the class library and validates the license key at runtime.

Parameters

nMaxFiles

An unsigned integer which specifies the maximum number of files which can be added to the queue. The constant INFINITE can be used to specify that there is no fixed limit to the number of files in the queue.

nTimeout

An unsigned integer which specifies the default timeout for all queued file transfers. If this value is zero, a reasonable default timeout period will be used. This timeout period is used when a file is added to the queue without providing a timeout period for that specific file transfer. If a timeout period is specified for a particular file, it will override this value.

dwOptions

An unsigned integer value which specifies one or more default options for all queued files. This parameter is constructed by using a bitwise operator and can be any of the options used with the **CHttpClient::Connect** method. If transfer options are specified for a particular file, it will override this value. The most common options are:

Value	Description
HTTP_OPTION_DEFAULT	Default options should be used for queued file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
HTTP_OPTION_FIREWALL	This option specifies queued transfers should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
HTTP_OPTION_SECURE	This option specifies queued file transfers should attempt to establish a secure connection with the server by default. This option is the same as specifying

	HTTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
HTTP_OPTION_SECURE_IMPLICIT	This option specifies queued file transfers should attempt to immediately establish secure TLS connection with the server by default. This option is typically used when connecting to a server on port 990, which is the default port number used for HTTPS.
HTTP_OPTION_SECURE_EXPLICIT	This option specifies queued file transfers should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
HTTP_OPTION_SECURE_SHELL	This option specifies queued file transfers should use the Secure Shell (SSH) protocol to establish the connection by default. This option will automatically be selected if the connection is established using port 22, the default port for SSH.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

If the ***nMaxFiles*** parameter is INFINITE, memory will be dynamically allocated on the process heap with no limit to the number of files which can be queued. If there is a logic error which causes the application to recursively add files to the queue, or repeatedly queue the same file, this can result in virtual memory being exhausted for the process. You can avoid this by specifying a reasonable maximum queue size, which will cause the **AddFile** method to fail if that limit is exceeded.

When called without any parameters, the constructor will create a queue with no file limit, default options and a default timeout period. The memory allocated for the queue will be released when the class destructor is invoked.

The constructor calls the **HttpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

~CHttpQueue, IsInitialized, CHttpClient::Connect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::~CHttpQueue

```
~CHttpQueue();
```

The **CHttpQueue** destructor releases resources allocated by the current instance of the **CHttpQueue** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CHttpQueue** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::AddFile Method

```
DWORD AddFile(
   DWORD dwQueueMode,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nTimeout
);

DWORD AddFile(
   DWORD dwQueueMode,
   LPCTSTR lpszHostName,
   UINT nHostPort,
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nTimeout
);
```

The **AddFile** method adds a new file to the queue with additional transfer options.

Parameters

dwQueueMode

An integer value which specifies if the file should be downloaded from the server or uploaded to the server. This parameter must be one of the following options and cannot have a value of zero.

Constant	Description
HTTP_QUEUE_DOWNLOAD	The file should be queued for downloading from the server to the local system.
HTTP_QUEUE_UPLOAD	The file should be queued for uploading from the local system to the server.

lpszHostName

The name of the server to establish a connection with. This may be a fully-qualified domain name or an IP address. If this parameter is NULL or points to an empty string, the ***lpszRemoteFile*** parameter must specify a complete URL which identifies the server as well as the path to the remote file.

nHostPort

The port number which should be used when establishing the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If this parameter is zero and a URL is specified with the ***lpszRemoteFile*** parameter, the port number included in the URL will be used as the default value.

nAuthType

An unsigned integer value which specifies the method to be used when authenticating the client. The following values are supported:

Value	Description
HTTP_AUTH_NONE	No client authentication should be performed. The ***lpszUserName*** and ***lpszPassword*** parameters are ignored and current authentication settings are cleared.
HTTP_AUTH_BASIC	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the connection. If this parameter is NULL or an empty string, no client credentials are provided. If the authentication type is HTTP_AUTH_BEARER, this parameter is ignored.

lpszPassword

A null-terminated string which specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified. If the authentication type is HTTP_AUTH_BEARER, this value must be the bearer token issued by the service provider.

lpszLocalFile

A null-terminated string which specifies the name of the local file to be queued for transfer. This parameter cannot be NULL or point to an empty string. If the file is being queued for an upload, the file must exist on the local system or the method will fail. If the file is being downloaded, it will be created or replaced on the local system.

lpszRemoteFile

A null-terminated string which specifies the name of the file on the server. If the file is being downloaded, the file must exist on the server or the transfer will fail. If the file is being uploaded, the file will be created or replaced on the server. This parameter may specify a complete URL. Note that any values you specify as arguments to this method will override the values specified in the URL. If you want to use a URL as the remote file name, the ***lpszHostName*** parameter should be NULL and the ***nHostPort*** parameter should be zero.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator and can be any of the options used with the **HttpClient::Connect** method. If this value is zero, the default options for the queue will be used. The most common options are:

Value	Description
HTTP_OPTION_DEFAULT	Default options should be used for the queued transfer. All standard headers will be included with the request and redirected resources will return an error, allowing the application to determine if it should request the resource from

	the new location or inform the user the resource location has changed.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.

nTimeout

The number of seconds to wait for a response from the server before failing the file transfer. If this value is zero, the default timeout period for the queue will be used.

lpvReserved

A reserved parameter which should always be NULL.

Return Value

If the method succeeds, the return value is a unique identifier which corresponds to the queued file. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will add a file to the current transfer queue. If successful, the value returned is a unique integer which identifies that file in the queue. To obtain information about the queued file, you can call the **GetFile** method and provide this return value as the ***dwFileId*** parameter. To remove a file from the queue, use the **RemoveFile** method.

Files can only be added to the queue when in an idle or paused state. If you attempt to add a file to the queue while the queue manager is transferring files, the method will fail. To determine the current state of the queue, call the **GetStatus** method

There are no fixed limits to the number of files which can be queued for transfer. To release the memory allocated for the queue, call the **Clear** method. To determine the current status of the queue, including how many file transfers have been queued and how many are pending completion, call the **GetStatus** method.

If the local file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the

ExpandEnvironmentStrings function in the Windows API. Note that environment variable names are not case-sensitive.

This method will not immediately begin the process of uploading or downloading the file. It will only add the transfer request to the current queue. The file transfers are managed by a background worker thread which is created when the **Start** method is called.

Example

```
// Create a new queue
CHttpQueue pQueue = new CHttpQueue();

if (!pQueue->IsInitialized())
    return;

// Add a file to the queue
DWORD dwFileId = pQueue->AddFile(
                     HTTP_QUEUE_DOWNLOAD,
                     lpszHostName,
                     HTTP_PORT_DEFAULT,
                     lpszUserName,
                     lpszPassword,
                     lpszLocalFile,
                     lpszRemoteFile);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (pQueue->GetFile(dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (pQueue->Start())
{
    pQueue->Wait(INFINITE);
    pQueue->Stop();
}

// Remove all files from the queue
```

```
delete pQueue;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CHttpQueue, ~CHttpQueue, FindFile, GetFile, GetFirstFile, GetNextFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::AttachHandle Method

```
VOID AttachHandle(
   HQUEUE hQueue
);
```

The **AttachHandle** method attaches the specified queue handle to the current instance of the class.

Parameters

hQueue

A handle to the queue.

Return Value

None.

Remarks

This method is used to attach a queue handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that queue.

If a handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous queue from being destroyed, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hQueue*** parameter is presumed to be a valid queue handle and no checks are performed to ensure that the handle is valid. Specifying an invalid queue handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Cancel Method

```
BOOL Cancel(
   DWORD dwFileId
);
```

The **Cancel** method cancels the specified file transfer.

Parameters

dwFileId

An optional unsigned integer value which uniquely identifies the file in the queue. If this parameter is omitted or the value is zero, the method will cancel the current file transfer in progress. If queue transfers have been suspended, this parameter cannot be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

When this method is called, the queued file transfer may not immediately stop. An internal flag is set which causes the file transfer to exit with an error and the queue manager will begin processing the next file in the queue. If the queue is in an idle state, this method will fail.

It is permitted to call the **Cancel** method from within a queue event handler to cancel the current file transfer. In this case, the *dwFileId* parameter should be zero. If you specify a file which has not been transferred, it will be flagged as canceled and skipped by the queue manager when processing the queue. If you specify a file which has already been processed, this method will fail.

A canceled file transfer is also considered a failed transfer. When you call **GetStatus** after a queued file transfer is canceled, the *dwFailedFiles* member of the HTTPQUEUESTATUS structure will be incremented to reflect this change. You can determine the status of an individual file transfer by calling the **GetFile** method and checking the value of the *dwQueueFlags* member of the HTTPQUEUEDFILE structure.

The **Reset** method can be used to reset the state of previously canceled transfers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Cancel, GetFile, GetStatus, Reset, Resume, Start, Suspend, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Clear Method

```
BOOL Clear();
```

The **Clear** method removes all files from the queue.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

This method can only be called when the queue is in an idle state. An error will be returned if the method is called while the queue manager is paused or actively transferring files in the queue. To determine the current state of the queue, call the **GetStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Cancel, GetStatus, Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::DetachHandle Method

```
HQUEUE DetachHandle();
```

The **DetachHandle** method detaches the queue handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the queue handle associated with the current instance of the class object. If an error occurs, the value INVALID_QUEUE will be returned.

Remarks

This method is used to detach a queue handle created by the class for use with the SocketTools API. Once the handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads. This option is always selected for the **CHttpQueue** class.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::EnumFiles Method

```
LONG EnumFiles(
   LPCTSTR lpszFileMask,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPDWORD lpFileList,
   LONG nMaxFiles
);
```

The **EnumFiles** method returns a list of files in the current transfer queue.

Parameters

lpszFileMask

A null-terminated string which specifies a file name which can include wildcards. Only those files which match this value will be enumerated. The character '?' will match against any single character, and '*' will match any number of characters. If this parameter is NULL or points to an empty string, all eligible files in the queue will be matched.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. Only those files which match the queue mode will be enumerated. This is a bitmask which may be one or more of the following values:

Constant	Description
HTTP_QUEUE_ALL	Match the file name to all files in the queue.
HTTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
HTTP_QUEUE_UPLOAD	Match file names which are queued for upload.
HTTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the enumeration. If this value is zero, no files will be excluded; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_QUEUE_FLAG_NONE	No files should be excluded.
HTTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
HTTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
HTTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **Cancel** method has been called and a queued file is in the process of

	being uploaded or downloaded.

lpFileList

A pointer to an array of unsigned integer values which will contain the unique file identifiers for each matching file in the queue. This parameter must specify an array large enough to store all of the file identifiers, otherwise the method will fail with a `ST_ERROR_BUFFER_TOO_SMALL` error. If this parameter is NULL, the method will return the number of matching files.

nMaxFiles

An integer value which specifies the maximum number of file identifiers which can be copied into the *lpFileList* array. If the *lpFileList* parameter is not NULL, this value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of matching files. If the queue is empty or there are no matching files, this method will return zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method populates an array of unique file identifiers which can be used to obtain information about the files in the current queue. The values returned in the *lpFileList* array can be used in conjunction with the **GetFile** method. This provides an alternative to using the **GetFirstFile** and **GetNextFile** methods.

The application should not make any assumptions about the value of the file identifiers returned by this method. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

Example

```
// Get the total number of files in the queue
LONG nFiles = pQueue->GetCount();

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];
    nFiles = pQueue->EnumFiles(NULL, HTTP_QUEUE_ALL, HTTP_QUEUE_FLAG_NONE,
pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
        HTTPQUEUEDFILE queuedFile;

        if (pQueue->GetFile(pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;
```

```
        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, FindFile, GetFile, GetFirstFile, GetNextFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::FindFile Method

```
DWORD FindFile(
   LPCTSTR lpszFileName,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles
);
```

The **FindFile** method finds a file in the transfer queue which matches the search criteria.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file to search for. The string may contain the wildcard character '?' to match against any single character, and '*' will match any number of characters. This parameter cannot be NULL or point to an empty string.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. This is a bitmask which may be one or more of the following values:

Constant	Description
HTTP_QUEUE_ALL	Match the file name to all files in the queue.
HTTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
HTTP_QUEUE_UPLOAD	Match file names which are queued for upload.
HTTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_QUEUE_FLAG_NONE	No files should be excluded from the search.
HTTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
HTTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
HTTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being uploaded or downloaded.

Return Value

If the method succeeds, the return value is a unique identifier for the file. To obtain information about the file, call the **GetFile** method. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns a unique file identifier for the first file which matches the specified file name in the queue. If the *lpszFileName* parameter includes wildcard characters, this method will return the first file in the queue which matches the name. File name matches are not case-sensitive, even when matching against remote file names on servers which use case-sensitive naming, such as UNIX based servers.

To perform more complex searches based on the file name, or to find multiple files, you can use the **GetFirstFile** and **GetNextFile** methods to iterate through all queued files.

Example

```
// Find a local file in the download queue
DWORD dwFileId = pQueue->FindFile(lpszFileName,
                                  HTTP_QUEUE_DOWNLOAD | HTTP_QUEUE_LOCAL,
                                  HTTP_QUEUE_FLAG_NONE);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (pQueue->GetFile(dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, EnumFiles, GetFile, GetFirstFile, GetNextFile, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetClient Method

```
HCLIENT GetClient();
```

The **GetClient** method returns a handle to the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value a handle to a client session. If the method fails, it will return INVALID_CLIENT. To get extended error information, call **GetLastError**.

Remarks

The **GetClient** method returns the handle to a client session which may be used with other methods. A valid handle will only be returned if the queue manager is currently uploading or downloading a file. If the queue is active but there are no file transfers in progress at the time this method is called, it will fail and set the last error code to `ST_ERROR_NO_QUEUED_TRANSFER`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpQueue, ~CHttpQueue, GetTransferStatus, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetCount Method

```
LONG GetCount();
```

The **GetCount** method returns the total number of files which have been queued for transfer.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of files in the queue. If the method fails, it will return HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpQueue, ~CHttpQueue, EnumFiles, GetStatus, GetFile, GetFirstFile, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetFile Method

```
BOOL GetFile(
   LPCTSTR lpszFileName,
   LPHTTPQUEUEDFILE lpFileInfo
);

BOOL GetFile(
   DWORD dwFileId,
   LPHTTPQUEUEDFILE lpFileInfo
);
```

The **GetFile** method returns information about the specified file in the transfer queue.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file in the queue. The name should always be the name of the remote file to search for.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

lpFileInfo

A pointer to a HTTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the file identifier is not valid or the file has been removed from the queue, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The file identifier can be obtained by several different methods, including **EnumFiles** and **GetStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

If you call the version of this method which accepts a file name, it will use the **FindFile** method to obtain the file ID for the first file name which matches a remote file in the queue. This approach is only recommended if you know that the queued file names will be unique.

Example

```
// Get the number of files in the queue
LONG nFiles = pQueue->GetCount();

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];

    nFiles = pQueue->EnumFiles(pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
```

```
        HTTPQUEUEDFILE queuedFile;

        if (pQueue->GetFile(pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumFiles, GetStatus, GetFile, GetFirstFile, GetNextFile, GetStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetFirstFile Method

```
DWORD GetFirstFile(
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPHTTPQUEUEDFILE lpFileInfo
);
```

The **HttpGetFirstFile** method returns information about the first file in the transfer queue.

Parameters

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. It may be one of the following values:

Constant	Description
HTTP_QUEUE_ALL	Return information about all files in the queue.
HTTP_QUEUE_DOWNLOAD	Return information for files which are queued for download.
HTTP_QUEUE_UPLOAD	Return information for files which are queued for upload.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_QUEUE_FLAG_NONE	No files should be excluded.
HTTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
HTTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
HTTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being uploaded or downloaded.

lpFileInfo

A pointer to a HTTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no queued files which match the specified parameters, or

the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns information about the first file in the current transfer queue. It is used in conjunction with the **GetNextFile** method to obtain information about all queued files.

Example

```
// List all files in a file transfer queue
HTTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = pQueue->GetFirstFile(HTTP_QUEUE_ALL, HTTP_QUEUE_FLAG_NONE, &queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case HTTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case HTTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
    }

    dwFileId = pQueue->GetNextFile(&queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, EnumFiles, FindFile, GetFile, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetHandle Method

```
HQUEUE GetHandle();
```

The **GetHandle** method returns the queue handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the queue handle associated with the current instance of the class object. If an error occurs, the value INVALID_QUEUE will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetNextFile Method

```
DWORD GetNextFile(
   LPHTTPQUEUEDFILE lpFileInfo
);
```

The **GetNextFile** method returns information about the next file in the transfer queue.

Parameters

lpFileInfo

A pointer to a HTTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no additional queued files which match the exclusion criteria specified with the call to **GetFirstFile**, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns information about the next file in the current transfer queue after an initial call to **GetFirstFile**. To obtain a list of all matching files in the queue, call this method repeatedly until it returns a value of zero. When information about the last file in the queue has been returned, the last error code will be set to `ST_ERROR_END_OF_QUEUE`.

Example

```
// List all files in a file transfer queue
HTTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = pQueue->GetFirstFile(HTTP_QUEUE_ALL, HTTP_QUEUE_FLAG_NONE, &queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case HTTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case HTTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
    }

    dwFileId = pQueue->GetNextFile(&queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, EnumFiles, FindFile, GetFile, GetFirstFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetStatus Method

```
INT GetStatus(
   LPHTTPQUEUESTATUS lpQueueStatus
);
```

The **GetStatus** method returns information about the specified file in the transfer queue.

Parameters

lpQueueStatus

A pointer to a HTTPQUEUESTATUS structure which will contain information about the current state of the file transfer queue. If this parameter is NULL, the method will ignore the parameter and only return the current status of the queue.

Return Value

If the method succeeds, the return value is the current queue status. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**. The following status values may be returned:

Value	Description
HTTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before **HttpStartQueue** has been called or after **HttpStopQueue** has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
HTTP_QUEUE_STATUS_ACTIVE	The queue manager is active and files are currently being uploaded or downloaded.
HTTP_QUEUE_STATUS_PAUSED	The queue manager is active although file transfers are currently paused. The queue enters this state after the **HttpSuspendQueue** method is called and resumes file transfers after the **HttpResumeQueue** method is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpQueue, ~CHttpQueue, Cancel, Reset, Resume, Start, Suspend, Stop, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::GetTransferStatus Method

```
INT GetTransferStatus(
   LPHTTPTRANSFERSTATUS lpStatus
);

INT GetTransferStatus(
   LPHTTPTRANSFERSTATUSEX lpStatus
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an HTTPTRANSFERSTATUS or HTTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. This method would typically be called from within a callback function in response to an HTTP_EVENT_PROGRESS event notification.

The *dwBytesTotal* and *dwBytesCopied* members of the **HTTPTRANSFERSTATUSEX** structure are declared as unsigned 64-bit integers rather than 32-bit integers. To obtain accurate file transfer information, this extended version of the structure should be used with files that are larger than 4GiB.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CHttpQueue, ~CHttpQueue, HttpEventProc, GetClient, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::IsEmpty Method

```
BOOL IsEmpty();
```

The **IsEmpty** method determines if the current queue is empty.

Parameters

None.

Return Value

If no files have been added to the queue, the return value is non-zero. If one or more files have been added to the queue, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AddFile Clear GetFile, IsIdle, RemoveFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::IsIdle Method

```
BOOL IsIdle();
```

The **IsIdle** method determines if the queue is idle.

Parameters

None.

Return Value

If the queue is idle and no files are being transferred, this method returns a non-zero value. If the queue manager is currently transferring files, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AddFile Cancel, Clear IsEmpty, Reset, Resume, Start, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpQueue, ~CHttpQueue, Clear GetClient, GetHandle Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::RemoveFile Method

```
BOOL RemoveFile(
   DWORD dwFileId
);

BOOL RemoveFile(
   LPCTSTR lpszFileName
);
```

The **RemoveFile** method removes the specified file from the queue.

Parameters

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

lpszFileName

A null-terminated string which specifies the name of the file to remove. If this version of the method is called, this parameter cannot be NULL or point to an empty string. The file name may include wildcard characters if you wish to remove multiple files. The character '?' will match against any single character, and '*' will match any number of characters.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

Files can only be removed from the queue when in an idle or paused state. If you attempt to remove a file while the queue manager is in the process of uploading or download files, the method will fail.

The file identifier can be obtained by several different methods, including **EnumFiles** and **GetStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

If you call this method using a file name, the **EnumFiles** method will be used to enumerate all remote file names which match and remove them from the queue. File name matches are not case-sensitive, even when matching against file names on servers which use case-sensitive naming, such as UNIX based servers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AddFile Clear IsEmpty, Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Reset Method

```
BOOL Reset(
   DWORD dwOptions
);
```

The **Reset** method resets the state of all file transfers in the current queue.

Parameters

dwOptions

An integer value which specifies which files should be reset in the queue. It may be one of the following values:

Value	Description
HTTP_QUEUE_RESET_ALL	All files in the queue should be reset to their initial state.
HTTP_QUEUE_RESET_COMPLETED	All files in the queue which have been successfully transferred will be reset to their initial state.
HTTP_QUEUE_RESET_FAILED	All files in the queue which were not transferred will be reset to their initial state.
HTTP_QUEUE_RESET_CANCELED	All files in the queue which were canceled will be reset to their initial state.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Start** method is called multiple times, it will not attempt to transfer files which have already been successfully copied, and it will not attempt to re-transfer files if the previous transfer failed. This method resets queued files back to their initial state, prior to when the queue manager attempted to perform the transfer. The most common use would be to call the method with the HTTP_QUEUE_RESET_FAILED mode and then call **Start** to retry failed or canceled file transfers.

Because canceled file transfers are also considered failed transfers, specifying HTTP_QUEUE_RESET_FAILED will reset queued files which either encountered an error during the transfer or were explicitly canceled by calling the **Cancel** method. The HTTP_QUEUE_RESET_CANCELED option will only reset the state of queued file transfers which were canceled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

AddFile Cancel, Clear, IsEmpty, Resume, Start, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Resume Method

```
BOOL Resume();
```

The **Resume** method resumes transfers in the queue.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

This method signals the queue manager to resume processing files in the transfer queue after the **Suspend** method has been called. This method will fail if the queue is in an idle (stopped) state, and will be ignored if the queue manager is already transferring files. The **GetStatus** method can be called to obtain the current status of the queue.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Cancel, Clear, GetStatus, Start, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Failure is typically indicated by a return value such as FALSE, NULL, INVALID_QUEUE or HTTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by calling the **GetLastError** method. The use of **GetLastError** is optional; an application can use the method to determine the specific reason for a failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** function in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Start Method

```
BOOL Start(
   DWORD dwQueueMode,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **Start** method begins transferring files in the queue.

Parameters

dwQueueMode

An unsigned integer which specifies which files in the queue should be transferred. It may be one of the following values:

Constant	Description
HTTP_QUEUE_ALL	All files in the queue should be transferred.
HTTP_QUEUE_DOWNLOAD	Transfer only those files which have been queued for download.
HTTP_QUEUE_UPLOAD	Transfer only those files which have been queued for upload.

lpEventProc

Specifies the procedure-instance address of an application defined callback function. For more information about the callback method, see the description of the **HttpEventProc** callback method. If this parameter is NULL, there will be no callback notifications.

dwParam

A user-defined integer value that is passed to the callback method. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

Queued file transfers are performed asynchronously using a background worker thread. If you provide the address to an event callback function, that method will always be invoked in the context of the queue manager thread. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself and you should avoid calling any methods which could cause the thread to block. For example, you should not attempt to establish other network connections from within the event handler.

If your application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback method. Controls should only be modified by the same UI thread that created their window. One common approach to address this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the ***this*** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

The **Wait** method can be used to wait for the queue manager to complete processing the transfer queue.

Example

```
// Create a new queue
CHttpQueue *pQueue = new CHttpQueue();

// Add a file to the queue
pQueue->AddFile(lpszLocalFile, lpszRemoteFile);

// Start the queued file transfers and wait for it to complete
if (pQueue->Start(HTTP_QUEUE_ALL))
{
    pQueue->Wait(INFINITE);
    pQueue->Stop();
}

// Remove all files from the queue
delete pQueue;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CHttpQueue, ~CHttpQueue, Cancel, Clear, GetStatus, Resume, Stop, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Stop Method

```
BOOL Stop();
```

The **Stop** method stops transferring queued files.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

When the **Stop** method is called and there is a file transfer in progress, it will not immediately stop the upload or download of the file. Instead, the queue manager is signaled to stop processing additional files in the queue after the transfer has completed. To wait for the current transfer to complete, call the **Wait** method.

It is permitted to call the **Stop** method from within a queue event handler. In this case, the current file transfer will complete and the queue manager will terminate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Cancel, Clear, GetStatus, Resume, Start, Suspend, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Suspend Method

```
BOOL Suspend();
```

The **Suspend** method pauses all file transfers for the queue.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

If there is a file transfer in progress when this method is called, the transfer will complete normally. The queue manager will enter a suspended state after the transfer has completed and before it begins processing the next file in the queue.

It is permitted to call the **Suspend** method from within a queue event handler. In this case, the current file transfer will complete and the queue manager will stop processing additional files in the queue until it is resumed or stopped.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Cancel, Clear, GetStatus, Resume, Start, Stop, Wait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpQueue::Wait Method

```
BOOL WINAPI HttpWaitForQueue(
   DWORD dwMilliseconds,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);

BOOL WINAPI HttpWaitForQueue(
   DWORD dwMilliseconds
);
```

The **Wait** method waits for the specified queue to complete the file transfers.

Parameters

dwMilliseconds

An unsigned integer value which specifies the number of milliseconds to wait for the queue to complete processing. If this value is zero, the method will return immediately. If this value is INFINITE (0xFFFFFFFF), the method will block indefinitely until all files in the queue have been processed.

lpdwElapsed

A pointer to an unsigned integer which will contain the number of milliseconds the queue manager has been active when the method returns. If this information is not needed, this parameter can omitted or it can be NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

lpdwError

A pointer to an unsigned integer which will contain the last error code set if a file transfer has failed or was canceled. If this information is not needed, this parameter can be omitted or NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

Return Value

If the method succeeds, the return value is non-zero, which means the queue manager has completed transferring all queued files. If the method fails, or the timeout period elapses, it will return zero. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until either the queue manager has completed processing all files in the queue, or until the specified number of milliseconds have elapsed. It is important to note that Windows messages will not be processed during this time. If you call this method within the main UI thread, it can potentially cause the application to become non-responsive. To determine the current state of the queue without blocking the current thread, call the **GetStatus** method.

If the *dwMilliseconds* parameter is non-zero and file transfers have not completed within the specified amount of time, the method will return zero and the last error code will be set to `ST_ERROR_OPERATION_TIMEOUT`.

If an event callback function has been specified when starting the queue, you should never call this method within the event handler. Attempting to do so can potentially create a situation which will cause the application to become non-responsive. The method will return zero if it determines it's being called within the context of the queue manager thread and will set the last error code to

`ST_ERROR_THREAD_DEADLOCK`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

Cancel, Clear, GetStatus, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Queue Data Structures

- HTTPQUEUEDFILE
- HTTPQUEUESTATUS
- HTTPTRANSFERSTATUS
- HTTPTRANSFERSTATUSEX
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPQUEUEDFILE

This structure is used by the **GetFile**, **GetFirstFile** and **GetNextFile** methods to return information about a file in the transfer queue.

```
typedef struct _HTTPQUEUEDFILE
{
    DWORD          dwFileId;
    DWORD          dwQueueMode;
    DWORD          dwQueueFlags;
    DWORD          dwTimeElapsed;
    DWORD          dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szLocalFile[HTTP_MAXFILENAMELEN];
    TCHAR          szRemoteFile[HTTP_MAXURLPATHLEN];
} HTTPQUEUEDFILE, *LPHTTPQUEUEDFILE;
```

Members

dwFileId

An unsigned integer which specifies a unique identifier for the queued file. The application should not make any assumptions about the value of the file identifiers. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

dwQueueMode

An unsigned integer value which specifies how the file was queued for transfer. It may be one of the following values:

Constant	Description
HTTP_QUEUE_DOWNLOAD	The file was queued for download.
HTTP_QUEUE_UPLOAD	The file was queued for upload.

dwQueueFlags

An unsigned integer which specifies one or more bit flags which provides information about the status of the file transfer. It may be one or more of the following values:

Constant	Description
HTTP_QUEUE_FLAG_NONE	The file is pending transfer in queue.
HTTP_QUEUE_FLAG_COMPLETED	The file has been transferred successfully. If this flag is set, no errors were encountered during the upload or download.
HTTP_QUEUE_FLAG_FAILED	The file transfer failed. If this flag is set, the ***dwError*** member of this structure will contain the error code associated with the failed transfer.
HTTP_QUEUE_FLAG_CANCELED	The file transfer was canceled. This flag is only set when the **Cancel** method has been called and a queued file is in the process of being uploaded or downloaded.

dwTimeElapsed

An unsigned integer which specifies the number of milliseconds required to complete the transfer. This value will be zero unless the file has been transferred successfully.

dwLastError

An unsigned integer which specifies the last error code for a failed transfer. If the file was transferred successfully, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the file transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

szLocalFile

A null-terminated string which specifies the full path to the local file being transferred.

szRemoteFile

A null-terminated string which specifies the complete URL to the remote file being transferred.

Remarks

It is possible that the ***szLocalFile*** and ***szRemoteFile*** structure members will be different than the values passed to the **AddFile** method. Those values are normalized, with any relative paths converted to absolute paths. Internationalized domain names will be encoded and the URL paths will be collapsed, removing any extraneous path information. For example, if the remote file name is specified as `http://http.server.tld/folder1/../folder2/filename.txt` it would be normalized as `http://http.server.tld/folder2/filename.txt`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddFile, GetFile, GetFirstFile, GetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPQUEUESTATUS

This structure is used by the **GetStatus** method to return information about the current status of the file transfer queue.

```
typedef struct _HTTPQUEUESTATUS
{
    DWORD dwStatus;
    DWORD dwThreadId;
    DWORD dwQueuedFiles;
    DWORD dwPendingFiles;
    DWORD dwCopiedFiles;
    DWORD dwFailedFiles;
    DWORD dwTimeElapsed;
    DWORD dwCurrentFile;
    DWORD dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    ULARGE_INTEGER uiBytesTotal;
} HTTPQUEUESTATUS, *LPHTTPQUEUESTATUS;
```

Members

dwStatus

An unsigned integer which specifies the current status of the queue. It can be one of the following values:

Value	Description
HTTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before the **Start** method has been called or after the **Stop** method has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
HTTP_QUEUE_STATUS_ACTIVE	Files in the queue are being uploaded or downloaded. The *dwCurrentFile* member of this structure identifies the file which is currently being transferred.
HTTP_QUEUE_STATUS_PAUSED	File transfers are currently paused. The queue enters this state after the **Suspend** method is called and resumes file transfers after the **Resume** method is called.

dwThreadId

An unsigned integer which specifies the thread ID for the queue manager which is performing the file transfers. All file transfers are performed asynchronously in a background worker thread. If this structure member is zero, the queue is idle and not performing file transfers. The application should not use the thread ID to obtain a handle to suspend or terminate the thread. This can potentially result in unexpected behavior or instability within the application.

dwQueuedFiles

An unsigned integer which specifies the total number of queued files. This value includes pending and completed file transfers.

dwPendingFiles

An unsigned integer which specifies the number of files which are queued to be transferred.

dwCopiedFiles

An unsigned integer which specifies the number of files which have been successfully transferred. This value reflects the total number of files which have the HTTP_QUEUE_FLAG_COMPLETED status.

dwFailedFiles

An unsigned integer which specifies the number of files which have failed to transfer or the file transfer has been canceled. This value reflects the total number of files which have the HTTP_QUEUE_FLAG_FAILED status.

dwTimeElapsed

An unsigned integer which specifies the total amount of time, in milliseconds, the queue manager has been active performing a file transfer. If the queue is idle, this value will reflect the total run time for the previously active queue. A value of zero indicates the queue was never active or the queue state has been reset with a call to the **Reset** method.

dwCurrentFile

An unsigned integer which specifies the unique identifier for the current file being transferred. If no file transfer is in progress, this member will have a value of zero. If the value is non-zero, it can be passed to the **GetFile** method to obtain information about the queued file.

dwLastError

An unsigned integer which specifies the last error code for a failed file transfer. If there have been no errors, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the current or last transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

uiBytesTotal

A value which specifies the total number of bytes which were copied during the current queue run. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

GetFile, GetStatus, Cancel, Reset, Resume, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPTRANSFERSTATUS

This structure is used by the **GetTransferStatus** method to return information about a data transfer in progress.

```
typedef struct _HTTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} HTTPTRANSFERSTATUS, *LPHTTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

GetTransferStatus, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPTRANSFERSTATUSEX

This structure is used by the **GetTransferStatus** method to return information about a data transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _HTTPTRANSFERSTATUSEX
{
   ULARGE_INTEGER uiBytesTotal;
   ULARGE_INTEGER uiBytesCopied;
   DWORD          dwBytesPerSecond;
   DWORD          dwTimeElapsed;
   DWORD          dwTimeEstimated;
} HTTPTRANSFERSTATUSEX, *LPHTTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

GetTransferStatus, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Server Class

Implements a server that enables the application to send and receive files using the Hypertext Transfer Protocol.

Reference

- Data Members
- Class Methods
- Event Handlers
- Data Structures
- Error Codes

Library Information

File Name	CSHTSV12.DLL
Version	12.0.3024.1286
LibID	25AA25CC-886B-4CFD-8089-96B70CAC2169
Import Library	CSHTSV12.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 3875

Overview

This class provides an interface for implementing an embedded, lightweight server that can be used to provide access to documents and other resources using the standard Hypertext Transfer Protocol. The server can accept connections from any standard web browser, third-party applications or programs developed using the SocketTools HTTP client API.

The application specifies an initial server configuration and then responds to events that are generated when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard HTTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server includes support for CGI scripting, virtual hosting, client authentication and the creation of virtual directories and files. The server also supports secure connections using TLS. Secure connections require a valid security certificate to be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer Public Data Members

Member Variables	Description
m_nCacheTime	The number of seconds until the client should consider a cached response to be stale
m_nExecTime	The maximum number of seconds that the server will permit an external command to execute
m_nIdleTime	The maximum number of seconds a client can be idle before the server terminates the session
m_nLogFormat	The format used by the server to log client activity
m_nLogLevel	The level of detail included in the server log file
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_nMaxPostSize	The maximum amount of data a client may submit to the server
m_nMaxRequests	The maximum number of requests a client may make per connection
m_dwOptions	Options specified when creating an instance of the server
m_dwStackSize	The initial size of the stack allocated for threads created by the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_nExecTime

```
UINT m_nExecTime;
```

The maximum number of seconds that the server will permit an external command to execute.

Remarks

The **m_nExecTime** data member is a public variable that specifies the maximum number of seconds that an external CGI program is permitted to run on the server. Programs are registered using the **RegisterProgram** method and are executed when the client sends a request for a resource that is associated with the program. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CHttpServer, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_nIdleTime

```
UINT m_nIdleTime;
```

The maximum number of seconds a client can be idle before the server terminates the session.

Remarks

The **m_nIdleTime** data member is a public variable that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 10 seconds and the maximum value is 300 seconds (5 minutes). This value is used to initialize the default idle timeout period for each client session. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CHttpServer, RegisterProgram, SetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_nLogFormat

```
UINT m_nLogFormat;
```

The format used when updating the server log file.

Remarks

The **m_nLogFormat** data member is a public variable that specifies the format of the log file that is created or updated by the server. It may be one of the following values:

Value	Description
HTTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

CHttpServer, GetLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_nLogLevel

```
UINT m_nLogLevel;
```

The level of detail included in the server log file.

Remarks

The **m_nLogLevel** data member is a public variable that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If the **m_nLogFormat** data member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format.

The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

CHttpServer, GetLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_nMaxClients

```
UINT m_nMaxClients;
```

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

CHttpServer, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_nMaxClientsPerAddress

```
UINT m_nMaxClientsPerAddress;
```

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

It is not recommended that you set the maximum clients per address below a value of 4. Lower values can negatively impact the performance of some clients, and may result in unexpected errors.

See Also

CHttpServer, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::m_dwOptions

```
DWORD m_dwOptions;
```

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see Server Option Constants. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CHttpServer, GetOptions, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer Methods

Class	Description
CHttpServer	Constructor which initializes the current instance of the class
~CHttpServer	Destructor which releases resources allocated by the class
Method	Description
AddVirtualHost	Add a new virtual host to the server virtual host table
AddVirtualHostAlias	Add an alternate host name for an existing virtual host
AddVirtualPath	Add a new virtual path for the specified host
AddVirtualUser	Add a new virtual user for the specified host
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
AuthenticateClient	Authenticate the client and assign access rights for the session
CheckVirtualPath	Determine if the client has permission to access the specified virtual path
DeleteAllClientHeaders	Delete all of the request or response headers for the specified client session
DeleteClientHeader	Delete a request or response header for the specified client session
DeleteVirtualHost	Delete a virtual host associated with the specified server
DeleteVirtualPath	Remove a virtual path from the specified host
DeleteVirtualUser	Delete a virtual user from the specified host
DetachHandle	Detach the server handle from the current instance of this class
DisableCommand	Disable a specific server command
DisableTrace	Disable the logging of network function calls
DisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
EnableClientAccess	Enable or disable access rights for the specified client session
EnableCommand	Enable a specific server command
EnableTrace	Enable logging of network function calls to a file
EnumClients	Returns a list of active client connections established with the server
GetActiveClient	Return the client ID for the active client session associated with the current thread
GetAddress	Return the IP address for the server
GetAllClientHeaders	Return all client request or response headers in a string buffer
GetClientAccess	Return the access rights that have been granted to the client session
GetClientAddress	Return the IP address of the specified client session
GetClientCredentials	Return the credentials for the specified client session
GetClientDirectory	Return the root document directory for a client session

GetClientHeader	Return the value of a request or response header for the specified client session
GetClientIdleTime	Return the idle timeout period for the specified client
GetClientLocalPath	Return the full local path for the specified virtual path
GetClientServer	Return the handle to the server that created the specified client session
GetClientThreadId	Returns the thread ID associated with the specified client session
GetClientUserName	Return the user name associated with the specified client session
GetClientVariable	Return the value of a CGI environment variable for the specified client
GetClientVirtualHost	Return the name of the virtual host the client used to establish the connection
GetClientVirtualHostId	Return the virtual host ID associated with the specified client session
GetClientVirtualPath	Return the virtual path for a local file on the server
GetCommandFile	Return the full path to the local file name or directory specified by the client
GetCommandLine	Return the complete command line issued by the client
GetCommandName	Return the name of the command that was issued by the client
GetCommandQuery	Return the query parameters included with the command
GetCommandResource	Return the path for the resource requested by the client
GetCommandResult	Return the result code and a description of the last command processed by the server
GetCommandUrl	Return the complete URL of the resource requested by the client
GetDirectory	Return the full path to the root directory assigned to the specified server
GetHandle	Return the server handle associated with the class instance
GetIdentity	Return the identity and version information for the specified server
GetLastError	Return information about the last server error that occurred
GetLogFile	Return the current log file format and full path for the file
GetMemoryUsage	Return the amount of memory allocated for the server and all client sessions
GetName	Return the host name assigned to the server or specified client session
GetOptions	Return the options specified for this instance of the server
GetPriority	Return the current priority assigned to the specified server
GetProgramExitCode	Return the exit code of the last program executed by the client
GetProgramName	Return the name of the CGI program executed by the client
GetProgramOutput	Return a copy of the standard output from the last program executed by the client
GetProgramText	Return a copy of the standard output from the last program in a string buffer
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetTransferInfo	Return information about the current file transfer
GetUuid	Return the UUID assigned to the specified server
GetVirtualHostId	Return the virtual host ID associated with the specified hostname

GetVirtualHostName	Return the hostname associated with the specified virtual host ID
IsActive	Determine if the server has been started
IsClientAuthenticated	Determine if the specified client session has been authenticated
IsCommandEnabled	Determine if the specified command is currently enabled or disabled
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the server is listening for client connections
PreProcessEvent	Filter server events before being processed by the default event handler
ReceiveRequest	Receive the request that was sent by the client to the server
RedirectRequest	Redirect the request from the client to another URL
RegisterHandler	Register a CGI program for use and associate it with a file name extension
RegisterProgram	Register a CGI program for use and associate it with a virtual path on the server
RenameServerLogFile	Rename or delete the current log file being updated by the server
RequireAuthentication	Send a response to the client indicating that authentication is required
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SendErrorResponse	Send a customized error response to the specified client
SendResponse	Send a response from the server to the specified client
SendResponseData	Send additional data to the client in response to a command
SetCertificate	Set the name of the certificate to be used with secure connections.
SetClientAccess	Change the access rights associated with the specified client session
SetClientHeader	Create or change the value of a request or response header for the client session
SetClientIdleTime	Change the idle timeout period for the specified client session
SetClientVariable	Create or change the value of a CGI environment variable for the specified client
SetCommandFile	Change the name of the local file or directory that is the target of the current command
SetDirectory	Specify the local directory that will be used as the server root directory
SetLastError	Set the last error code for the specified server session
SetLogFile	Change the current log format, level of detail and file name
SetIdentity	Change the identity and version information for the specified server
SetName	Change the hostname assigned to the server or specified client session
SetOptions	Change the options specified for this instance of the server
SetPriority	Change the priority assigned to the specified server
SetStackSize	Change the initial size of the stack allocated for threads created by the server
SetUuid	Assign a UUID to be associated with this instance of the server
Start	Start the server and begin accepting client connections

Stop	Stop the server and terminate all active client connections
Suspend	Suspend accepting client connections on the specified server
Throttle	Limit the number of active client connections, connections per address and connection rate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::~CHttpServer

```
~CHttpServer();
```

The **CHttpServer** destructor releases resources allocated by the current instance of the **CHttpServer** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CHttpServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CHttpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AddVirtualHost Method

```
UINT HttpAddVirtualHost(
   LPCTSTR lpszHostName,
   UINT nHostPort,
   LPCTSTR lpszDirectory
);
```

Add a new virtual host to the server virtual host table.

Parameters

lpszHostName

A null-terminated string which specifies the hostname that will be added to the virtual host table. This parameter must specify a valid hostname and cannot be a NULL pointer or a zero-length string.

nHostPort

An integer value which specifies the port number for the virtual host. This value must be zero or the same value as the original port number that the server was configured to use.

lpszDirectory

A null-terminated string which specifies the root document directory for the virtual host. If this parameter is NULL or a zero-length string, the virtual host will use the same root directory that was specified when the server was started. This parameter may contain environment variables enclosed in % symbols.

Return Value

If the method is successful, it will return a non-zero integer value that identifies the virtual host. If the method fails, it will return INVALID_VIRTUAL_HOST and the last error code will be updated to indicate the cause of the failure.

Remarks

Virtual hosting is a method for sharing multiple domain names on a single instance of a server. The client provides the server with the hostname that it has used to establish the connection, and that name is compared against a table of virtual hosts configured for the server. If the hostname matches a virtual host, the client will use the root directory and any virtual paths that have been assigned to that host.

When the server is first started, a default virtual host with an ID of zero is automatically created and is identified as VIRTUAL_HOST_DEFAULT. This virtual host uses the same hostname, port number and root directory that the server instance was created with. The application should treat all other host IDs as opaque values and never make assumptions about how they are allocated.

The *nHostPort* parameter should always be specified with a value of zero, or the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHostAlias, AddVirtualPath, DeleteVirtualHost, GetVirtualHostId, GetVirtualHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AddVirtualHostAlias Method

```
BOOL AddVirtualHostAlias(
   UINT nHostId,
   LPCTSTR lpszHostAlias
);
BOOL AddVirtualHostAlias(
   LPCTSTR lpszHostAlias
);
```

Add an alternate host name for an existing virtual host.

Parameters

nHostId

An integer value which identifies the virtual host. If this parameter is omitted, the alias will be assigned to the default server.

lpszHostAlias

A null-terminated string which specifies the alias for the virtual host. The alias must be a valid domain name that uniquely identifies the host. This parameter cannot be a NULL pointer or specify an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualHostAlias** method adds an alias for an existing virtual host. This enables a client to establish a connection using a number of different domain names which all reference the same virtual host. When the server responds to the client, it will identify itself with the primary domain name assigned to the virtual host rather than the alias provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, AddVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AddVirtualPath Method

```
BOOL AddVirtualPath(
   UINT nHostId,
   LPCTSTR lpszVirtualPath,
   LPCTSTR lpszLocalPath,
   DWORD dwFileAccess
);
```

Add a new virtual path for the specified host.

Parameters

nHostId

An integer value which identifies the virtual host. If this parameter is omitted, the default virtual host will be used.

lpszVirtualPath

A null-terminated string which specifies the virtual path that will be created. This parameter cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

lpszLocalPath

A null-terminated string which specifies the local directory or file name that the virtual path will be mapped to. This path must exist and can be no longer than MAX_PATH characters. This parameter cannot be a NULL pointer or an empty string.

dwFileAccess

An integer value which specifies the access clients will be given to the virtual path. For a list of file access permissions, see User and File Access Constants.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualPath** method maps a virtual path name to a directory or file name on the local system. Virtual paths are assigned to specific hosts and if multiple virtual hosts are created for the server, each can have its own virtual paths which map to different files. To create a virtual path for the default server, the ***nHostId*** parameter should be specified as VIRTUAL_HOST_DEFAULT.

It is recommended that the ***lpszLocalPath*** parameter always specify the full path to the local file or directory. If the path is relative, it will be considered relative to the current working directory for the process and expanded to its full path name. The local path can include environment variables surrounded by % symbols. For example, if the value %ProgramData% is included in the path, it will be expanded to the full path for the common application data folder. The local path cannot specify a Windows system folder or the root directory of a mounted drive volume.

The local file or directory does not need to located in the document root directory for the server or virtual host. It can specify any valid local path that the server process has the appropriate permissions to access. You should exercise caution when creating virtual paths to files or directories outside of the server root directory. If the ***lpszLocalPath*** parameter specifies a directory, clients will have access to that directory and all subdirectories using its virtual path.

If you wish to password protect the virtual file or directory, include the HTTP_ACCESS_PROTECTED flag in the file permissions. The default command handlers will recognize this flag and require that the client authenticate itself to grant access to the resource. If the server application implements a custom command handler, it is responsible for checking for the presence of this flag and perform the appropriate checks to ensure that the client session has been authenticated.

If the server was started in restricted mode, the client will be unable to access documents outside of the server root directory and its subdirectories. This restriction also applies to virtual paths that reference documents or other resources outside of the root directory. To allow a client to access a document outside of the server root directory, the **SetClientAccess** method should be used to grant the client HTTP_ACCESS_READ permission.

The **GetClientVirtualPath** method will return the virtual path that is associated with a local file or directory. The **GetClientLocalPath** method will return the full path to a local file or directory that is mapped to a virtual path.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, CheckVirtualPath, GetClientLocalPath, GetClientVirtualPath, DeleteVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AddVirtualUser Method

```
BOOL AddVirtualUser(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   DWORD dwUserAccess,
   LPCTSTR lpszDirectory
);

BOOL AddVirtualUser(
   UINT nHostId,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   DWORD dwUserAccess,
   LPCTSTR lpszDirectory
);
```

Add a new virtual user for the specified host.

Parameters

nHostId

An integer value which identifies the virtual host. If this parameter is omitted, the default virtual host will be used.

lpszUserName

A null-terminated string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the method will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A null-terminated string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the method will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see User and File Access Constants. If this parameter is omitted, default access rights will be assigned based on the server configuration.

lpszDirectory

A null-terminated string which specifies the local directory that is considered to be the virtual user's home directory. This path must exist and can be no longer than MAX_PATH characters. The maximum length of the local path is 260 characters. The directory cannot be located in a Windows system folder or the root directory of a mounted disk volume. If this parameter is omitted, the server root directory will be assigned as the user home directory.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid

host, or the username or password contain invalid characters, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualUser** method adds a virtual user that is associated with the specified virtual host. If a client attempts to access a protected document and provides credentials, the server will attempt to automatically authenticate the session by searching for virtual user with the same username and password. If a match is found, then the client session is assigned the same access permissions as the virtual user.

If the server is started with the HTTP_SERVER_MULTIUSER option, then documents in the virtual user's home directory can be accessed by specifying their username using a specially formatted request URL. For example, if a virtual user named "Thomas" is created, the documents in that user's home directory could be accessed as http://***servername***/~thomas/***document***.html

All files and subdirectories in the user's home directory are considered to be read-only. A client cannot create files in a user's home directory, even if they are authenticated as that user. In addition, CGI programs and scripts cannot be executed from a user's home directory.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this method is called with a username that already exists, that record is replaced with the values passed to this method.

The virtual users created by this method exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this method for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, DeleteVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AsyncNotify Method

```
BOOL AsyncNotify(
   HWND hWnd,
   UINT uMsg
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the ***hWnd*** parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the ***hWnd*** parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The ***wParam*** message parameter will contain the notification message and the ***lParam*** message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Value	Description
HTTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the **Start** method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the **Resume** method to resume

	accepting client connections. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the **Suspend** method. This notification message may be sent to the application multiple times over the lifetime of the server. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the **Restart** method. Note that the server socket handle provided by the ***lParam*** message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The ***lParam*** message parameter will specify the unique ID of the client that connected to the server.
HTTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the **Stop** method. The ***lParam*** message parameter will specify the unique ID of the client that disconnected from the server.
HTTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in ***lParam*** will reference the defunct server and cannot be used with other server methods. The ***lParam*** message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The HTTP_NOTIFY_CONNECT and HTTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the ***lParam*** message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **GetClientServer** method to obtain a handle to the server that created the client session. Note that at the time the application processes the HTTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: cshtsv12.lib

See Also

GetClientServer, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AttachHandle Method

```
VOID AttachHandle(
   HSERVER hServer
);
```

The **AttachHandle** method attaches the specified server handle to the current instance of the class.

Parameters

hServer

The handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle has already been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the ***hServer*** parameter is presumed to be a valid server handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid server handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::AuthenticateClient Method

```
BOOL AuthenticateClient(
   UINT nClientId,
   DWORD dwUserAccess
);
```

Authenticate the client and assign access rights for the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User and File Access Constants.

Return Value

If the client session could be authenticated, the return value is non-zero. If the client ID does not specify a valid client session, or the client has already been authenticated, this method will return zero.

Remarks

The **HttpAuthenticateClient** method is used to authenticate a specific client session, typically in response to an **OnAuthenticate** event that indicates a client has provided authentication credentials as part of the request for a document or other resource.

To enable the server to automatically authenticate a client session, use the **AddVirtualUser** method to add one or more virtual users. The server will search the list of virtual users for a match to the credentials provided by the client and will set the appropriate permissions for the session without requiring a event handler to manually authenticate the session using this method.

If the server was started with the HTTP_SERVER_LOCALUSER option and the client session is not authenticated using this method, the server will attempt to authenticate the client session using the local Windows user database. Although this option can be convenient because it does not require the implementation of an event handler for the **OnAuthenticate** event, it can be used by clients to attempt to discover valid usernames and passwords for the local system. It is recommended that you use the **AddVirtualUser** method to create virtual users rather than using the local user database.

It is recommended that most applications specify HTTP_ACCESS_DEFAULT as the ***dwUserAccess*** value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualUser, GetClientCredentials, GetClientDirectory, OnAuthenticate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::CheckVirtualPath Method

```
BOOL CheckVirtualPath(
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   DWORD dwFileAccess
);
```

Determine if the client has permission to access the specified virtual path.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A null-terminated string which specifies the virtual path that should be checked. This path must be absolute and cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

dwFileAccess

An unsigned integer value which specifies the access permissions that should be checked. For a list of file access permissions, see User and File Access Constants.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **CheckVirtualPath** method is used to determine if the client has permission to access the virtual file or directory, based on the value of the *dwFileAccess* parameter. For example, if the *dwFileAccess* parameter has the value HTTP_ACCESS_WRITE, this method will check if the client has write permission for the file or directory. The method will return a non-zero value if the client does have the requested permission, or zero if it does not.

Applications that implement their own custom handlers for standard HTTP commands should use this method to ensure that the client has the appropriate permissions to access the requested resource. Failure to check the access permissions for the client can result in the client being able to access restricted documents and other resources. It is recommended that most applications use the default command handlers.

To obtain the path to the local file or directory that the virtual path is mapped to, use the **GetClientLocalPath** method.

Example

```
CString strPathName;

// Get the current request URL path
INT cchPathName = pServer->GetCommandResource(nClientId, strPathName);

if (cchPathName == 0)
{
    pServer->SendErrorResponse(nClientId, 500);
```

```
    return;
}

// Check if the client has write access to that resource
BOOL bAllowed = pServer->CheckVirtualPath(
                    nClientId,
                    strPathName,
                    HTTP_ACCESS_WRITE);

if (!bAllowed)
{
    pServer->SendErrorResponse(nClientId, 403);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualPath, DeleteVirtualPath, GetClientLocalPath, GetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::CHttpServer Method

```
CHttpServer();
```

The **CHttpServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **HttpServerInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** method because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CHttpServer, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DeleteAllClientHeaders Method

```
BOOL DeleteAllClientHeaders(
   UINT nClientId,
   UINT nHeaderType
);
```

Delete all of the request or response headers for the specified client session.

Parameters

nClientId

Delete all of the request or response headers for the specified client session.

nHeaderType

Specifies the type of headers to delete. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that were provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete all of the response headers that were created by the server in response to a request made by the client. Response header values provide additional information to the client about the type of information that is being returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **DeleteAllClientHeaders** method is used to delete all of the request or response headers that were set as the result of the client issuing a request for a document or other resource. If this method is used to delete all of the response headers, the server will automatically generate a standard set of response headers when it returns the requested information to the client.

It is not necessary to call this method inside an **OnDisconnect** event handler to delete the header values that were set during the client session. This is done automatically when the client disconnects from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

DeleteClientHeader, SetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DeleteClientHeader Method

```
BOOL DeleteClientHeader(
   UINT nClientId,
   UINT nHeaderType,
   LPCTSTR lpszHeaderName
);
```

Delete a request or response header for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Delete a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A null-terminated string that specifies the name of the header that should be deleted. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **DeleteClientHeader** method will delete a request or response header for the specified client session. There are a number of required response headers that are always sent to a client and deleting the header using this method will cause the server to automatically generate a new default header value. You should not delete response header values unless you are certain of the impact that it would have on the normal operation of the client.

It is not necessary for you to delete a header value to change the value of an existing header. The **SetClientHeader** method will replace an existing header value with a new value, or create a new header if the header name does not already exist.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteAllClientHeaders, SetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DeleteVirtualHost Method

```
BOOL DeleteVirtualHost(
   UINT nHostId
);
```

Delete a virtual host associated with the specified server.

Parameters

nHostId

An integer value which identifies the virtual host.

Return Value

If the method succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **DeleteVirtualHost** method removes a virtual host that was created by a previous call to the **AddVirtualHost** method. All virtual paths and users associated with the specified host are no longer valid. It is not necessary to call this method to delete any of the virtual hosts prior to stopping the server. Part of the normal shutdown process is releasing the resources allocated for each virtual host that was added to the server.

This method cannot be used to delete the virtual host with an ID of zero, which is the default virtual host that is allocated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

AddVirtualHost, AddVirtualPath, AddVirtualUser DeleteVirtualPath, DeleteVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DeleteVirtualPath Method

```
BOOL DeleteVirtualPath(
   UINT nHostId,
   LPCTSTR lpszVirtualPath
);
```

Remove a virtual path from the specified host.

Parameters

nHostId

An integer value which identifies the virtual host.

lpszVirtualPath

A null-terminated string which specifies the virtual path that will be removed. This path must be absolute and cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This method removes a virtual path that was created by a previous call to the **AddVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, AddVirtualPath, GetClientLocalPath, GetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DeleteVirtualUser Method

```
BOOL DeleteVirtualUser(
   UINT nHostId,
   LPCTSTR lpszUserName
);
```

Remove a virtual user from the specified host.

Parameters

nHostId

An integer value which identifies the virtual host.

lpszUserName

A null-terminated string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, or the username does not exist, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This method removes a virtual user that was created by a previous call to the **AddVirtualUser** method. This method will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, AddVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DetachHandle Method

```
HSERVER DetachHandle();
```

The **DetachHandle** method detaches the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If there is no active server, the value INVALID_SERVER will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketTools API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DisableCommand Method

```
BOOL EnableCommand(
   LPCTSTR lpszCommand
);
```

Disable a specific server command.

Parameters

lpszCommand

A null-terminated string that specifies the name of the command to be disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the command is not recognized, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **DisableCommand** method is used to disable access to a specific command on the server, typically for security purposes. For example, the PUT command can be disabled, preventing any client from attempting to upload files directly to the server. The **IsCommandEnabled** method can be used to determine if a command is enabled or not.

The command name provided to this method must match the commands defined in RFC 2616 or related protocol standards. Refer to Hypertext Transfer Protocol Commands for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the GET and HEAD commands cannot be disabled. If you attempt to disable a required command, this method will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this method affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, use an event handler to filter the commands.

The OPTIONS and TRACE commands are disabled by default for all server instances and must be explicitly enabled using the **EnableCommand** method if you wish permit clients to use them. It is not recommended that you enable these commands if your server is going to be publicly accessible over the Internet. If the server started with the option HTTP_SERVER_READONLY, commands that can be used to create or modify files on the server will be disabled by default.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, EnableCommand, IsCommandEnabled

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DisableTrace Method

```
BOOL DisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::DisconnectClient Method

```
BOOL DisconnectClient(
   UINT nClientId
);
```

Close the control connection for the specified client and release the resources allocated for the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **DisconnectClient** method will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted. This performs the same operation as if the client sent the QUIT command to the server.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active file transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

After this method returns, the application should never use the same client ID with another method. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

Restart, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::EnableClientAccess Method

```
BOOL EnableClientAccess(
   UINT nClientId,
   DWORD dwUserAccess,
   BOOL bEnable
);
```

Enable or disable access rights for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see User and File Access Constants.

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **EnableClientAccess** method is used to enable or disable access to specific functionality by the client. The method can only change a single access right and cannot be used to enable or disable multiple access rights in a single method call. To change multiple user access rights for the client, use the **SetClientAccess** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

AuthenticateClient, GetClientAccess, SetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::EnableCommand Method

```
BOOL EnableCommand(
   LPCTSTR lpszCommand
);
```

Enable a specific server command.

Parameters

lpszCommand

A null-terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

bEnable

An integer value which specifies if the command should be enabled or disabled. If the value is non-zero, the command is enabled. If the value is zero, the command will be disabled.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **EnableCommand** method is used to enable access to a specific command on the server. The **IsCommandEnabled** method can be used to determine if a command is enabled or not. The command name provided to this method must match the commands defined in RFC 2616 or related protocol standards. Refer to Hypertext Transfer Protocol Commands for a complete list of server commands.

The OPTIONS and TRACE commands are disabled by default for all server instances and must be explicitly enabled using the **EnableCommand** method if you wish permit clients to use them. It is not recommended that you enable these commands if your server is going to be publicly accessible over the Internet. If the server started with the option HTTP_SERVER_READONLY, commands that can be used to create or modify files on the server will be disabled by default.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, DisableCommand, IsCommandEnabled

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the process level and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "`0x`". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::EnumClients Method

```
INT EnumClients(
   UINT *lpClients,
   INT nMaxClients
);
```

Return a list of active client connections established with the server.

Parameters

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the ***lpClients*** array. If the ***lpClients*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections, the method will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the method with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero.

Example

```
// Populate a listbox with all of the users connected to the server
pListBox->ResetContent();

INT nClients = pHttpServer->EnumClients();
if (nClients > 0)
{
    UINT *pIdList = new UINT[nClients];

    nClients = pHttpServer->EnumClients(pIdList, nClients);
    if (nClients == HTTP_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        CString strUserName;

        if (pHttpServer->GetClientUserName(pIdList[nIndex], strUserName))
            pListBox->AddString(strUserName);
    }

    // Free the memory allocated for the client IDs
```

```
    delete pIdList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetClientAddress, GetClientDirectory, GetClientUserName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetActiveClient Method

```
UINT GetActiveClient();
```

Return the client ID for the active client session associated with the current thread.

Parameters

None.

Return Value

If the method succeeds, the return value is the unique ID associated with the client session for the current thread. If there is no client session active on the current thread, the return value is zero.

Remarks

The **GetActiveClient** method is used to obtain the client ID associated with the current thread. This means this method will only return a client ID if it is called within an event handler or a method called by an event handler. If this method is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

EnumClients

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetAddress Method

```
INT GetAddress(
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetAddress(
   UINT nClientId,
   CString& strAddress
);
```

Return the IP address of the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A null-terminated string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

This method will return the IP address assigned to the specified server as a printable string. If the ***nClientId*** parameter has a value of zero, this method will return the IP address assigned to the local system. If the HTTP_SERVER_NATROUTER option was specified when the server was started, this method will return the external IP address assigned to the system. If the ***nClientId*** parameter specifies a valid client session, this method will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **GetClientAddress** method.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientAccess Method

```
BOOL GetClientAccess(
   UINT nClientId,
   DWORD& dwUserAccess
);
```

Return the access rights that have been granted to the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see User and File Access Constants.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **GetClientAccess** method is used to obtain all of the access rights that are currently granted to an authenticated client session. The **EnableClientAccess** method can be used to enable or disable specific permissions, and the **SetClientAccess** method can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;

// Check if the client has execute permission
if (pHttpServer->GetClientAccess(nClientId, dwUserAccess))
{
    if (dwUserAccess & HTTP_ACCESS_EXECUTE)
    {
        std::cout << "Client can execute programs and scripts\n";
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

AuthenticateClient, EnableClientAccess, SetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientAddress Method

```
INT GetClientAddress(
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetClientAddress(
   UINT nClientId,
   CString& strAddress
);
```

Return the IP address of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A null-terminated string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientCredentials Method

```
BOOL GetClientCredentials(
   UINT nClientId,
   LPHTTPCLIENTCREDENTIALS lpCredentials
);

BOOL GetClientCredentials(
   UINT nClientId,
   CString& strUserName,
   CString& strPassword
);
```

Return the user credentials for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **HTTPCLIENTCREDENTIALS** structure that will contain information about the user when the method returns. This parameter cannot be NULL.

strUserName

A string that will contain the user name when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

strPassword

A string that will contain the user password when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not requested authentication, this method will return zero.

Remarks

The **GetClientCredentials** method is used to obtain the username and password that was provided by the client when it requested authentication. Typically this method is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **AuthenticateClient** method to specify that the session has been authenticated.

If the default event handler is used, the **OnAuthenticate** method will be invoked with the user credentials passed to the handler as arguments.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, RequireAuthentication, OnAuthenticate, HTTPCLIENTCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientDirectory Method

```
INT GetClientDirectory(
   UINT nClientId,
   LPTSTR lpszDirectory,
   INT nMaxLength
);

INT GetClientDirectory(
   UINT nClientId,
   CString& strDirectory
);
```

Returns the root document directory for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string buffer that will contain the root document directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero.

Remarks

This method returns the full path to the root document directory for the specified client session. If no virtual hosts have been configured, then this value will be the same as the root directory assigned to the server when it was started. If the server has been configured with multiple virtual hosts, this function will return the path to the root directory associated with the hostname provided by the client.

To convert a full path to the virtual path for a specific client session, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientHeader Method

```
INT GetClientHeader(
   UINT nClientId,
   UINT nHeaderType,
   LPCTSTR lpszHeaderName,
   LPTSTR lpszHeaderValue,
   INT nMaxLength,
);
```

Return the value of a request or response header for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Return the value of a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Return the value of a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A null-terminated string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A buffer that will contain the header value, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the ***nMaxLength*** parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the header value buffer, including the terminating null character. If the ***lpszHeaderValue*** parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the length of the header value, not including the terminating null character. If the client ID does not specify a valid client session, or there is no header that matches the given name, the method will return zero. If the ***lpszHeaderValue*** parameter is not NULL and the buffer is not large enough to store the complete header value, the method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **GetClientHeader** method will return the value of a request or response header for the specified client session. If the *lpszHeaderName* value matches an existing header field, its value will be copied to the string buffer provided by the caller.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteClientHeader, GetAllClientHeaders, SetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientIdleTime Method

```
UINT GetClientIdleTime(
   UINT nClientId,
   UINT *lpnElapsed
);
```

Return the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

An optional pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL or omitted if this information is not required.

Return Value

If the method succeeds, the return value is client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetClientIdleTime** method will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed. The server will never disconnect a client that is in the process of sending or receiving data, regardless of the idle timeout period.

The default idle timeout period for a client session is 60 seconds, however the server can be configured to use a different value. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). An application can change the timeout period for a specific client session using the **SetClientIdleTime** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

SetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientLocalPath Method

```
INT GetClientLocalPath(
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   LPTSTR lpszLocalPath,
   INT nMaxLength,
);

INT GetClientLocalPath(
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   CString& strLocalPath
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A null-terminated string that specifies an virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A null-terminated string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **GetClientLocalPath** method takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the root directory for the client session.

To obtain the virtual path for a local file or directory, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientServer Method

```
HSERVER HttpGetClientServer(
    UINT nClientId
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SERVER. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return INVALID_SERVER indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: cshtsv12.lib

See Also

AsyncNotify

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientThreadId Method

```
DWORD GetClientThreadId(
   UINT nClientId
);
```

Returns the thread ID associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **DisconnectClient** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

EnumClients, GetActiveClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientUserName Method

```
INT GetClientUserName(
   UINT nClientId,
   LPTSTR lpszUserName,
   INT nMaxLength
);

INT GetClientUserName(
   UINT nClientId,
   CString& strUserName
);
```

Return the user name associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

strUserName

A null-terminated string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not authenticated itself, the method will return zero and the *lpszUserName* parameter will be set to an empty string.

The **IsClientAuthenticated** method can be used to determine if the client has provided credentials as part of the request made to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, GetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientVariable Method

```
INT GetClientVariable(
   UINT nClientId,
   LPCTSTR lpszName,
   LPTSTR lpszValue,
   INT nMaxLength,
);

INT GetClientVariable(
   UINT nClientId,
   LPCTSTR lpszName,
   CString& strValue
);
```

Return the value of a CGI environment variable for the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A null-terminated string that specifies the name of the environment variable. Variable names are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A buffer that will contain the value of the environment variable, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the ***nMaxLength*** parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the value buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the length of the environment variable value, not including the terminating null character. If the client ID does not specify a valid client session, or there is no environment variable that matches the given name, the method will return zero. If the ***lpszValue*** parameter is not NULL and the buffer is not large enough to store the complete header value, the method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **GetClientVariable** method will return the value of an environment variable that has been defined for the client. Each client session inherits a copy of the process environment block, which is then modified to define various environment variables that are used with CGI programs and scripts. The **SetClientVariable** method can be used to change existing environment variables or create new variables.

The standard CGI environment variables that are defined by the server are not created until the client

request has been processed. This means that environment variables such as REMOTE_ADDR and SERVER_NAME will not be defined inside an **OnConnect** event handler.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetClientVariable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientVirtualHost Method

```
INT GetClientVirtualHost(
   UINT nClientId,
   LPTSTR lpszHostName,
   INT nMaxLength
);

INT GetClientVirtualHost(
   UINT nClientId,
   CString& strHostName
);
```

Return the name of the virtual host the client used to establish the connection.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A null-terminated string buffer that will contain the virtual host name. The string buffer will be null-terminated and must be large enough to store the complete hostname. If this parameter is NULL, the method will only return the length of the current command in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszHostName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetClientVirtualHost** method is used to obtain the hostname that the client used to establish a connection with the server. This method is typically used within an event handler to determine the hostname associated with the request made by the client. It should not be called inside an **OnConnect** event handler because the virtual host has not been selected at that point. If the virtual hostname is not available at the time this method is called, the method will return zero and the last error code will be set to ST_ERROR_VIRTUAL_HOST_NOT_FOUND.

The **GetClientVirtualHostId** method can be used to obtain the virtual host ID associated with the hostname.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientVirtualHostId, GetCommandUrl, GetVirtualHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientVirtualHostId Method

```
UINT GetClientVirtualHostId(
    UINT nClientId
);
```

Return the virtual host ID associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An unsigned integer value which specifies the virtual host ID. If the method fails, the return value will be INVALID_VIRTUAL_HOST and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetClientVirtualHostId** method is used to obtain the virtual host ID associated with the hostname the client used to establish a connection with the server. This method should not be called inside an **OnConnect** event handler because the virtual host has not been selected at that point. If the virtual host ID is not available at the time this method is called, the method will return INVALID_VIRTUAL_HOST and the last error code will be set to ST_ERROR_VIRTUAL_HOST_NOT_FOUND.

The **GetClientVirtualHost** method can be used to obtain the hostname that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetClientVirtualHost, GetCommandUrl, GetVirtualHostId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientVirtualPath Method

```
INT GetClientVirtualPath(
   UINT nClientId,
   LPCTSTR lpszLocalPath,
   LPTSTR lpszVirtualPath,
   INT nMaxLength,
);

INT GetClientVirtualPath(
   UINT nClientId,
   LPCTSTR lpszLocalPath,
   CString& strVirtualPath
);
```

Return the virtual path for a local file on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A null-terminated string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A null-terminated string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is relative to the root directory for the specified client session. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the ***lpszLocalPath*** parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this method would return the virtual path to that directory as "/Documents/Research".

If the ***lpszLocalPath*** parameter specifies a file or directory outside of the server root directory, this method will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This method can only be used with authenticated clients. If the ***nClientId*** parameter specifies a client session that has not been authenticated, this method will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetClientLocalPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandFile Method

```
INT GetCommandFile(
   UINT nClientId,
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetCommandFile(
   UINT nClientId,
   CString& strFileName
);
```

Get the full path to a file name or directory specified by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null-terminated and must be large enough to store the complete file path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetCommandFile** method is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard HTTP command. For example, if the client sends the GET command to the server, this method will return the complete path to the local file that the client wants to retrieve. This method will only work with those standard commands that perform some action on a file or directory.

This method should always be used to obtain the file name for a command that performs a file or directory operation. It normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location. The **GetCommandUrl** method can be used to obtain the URL that was provided by the client.

To map a virtual path to a file or directory on the local system, use the **AddVirtualPath** method. To redirect a client to use a different URL to access the resource, use the **RedirectRequest** method.

The **SetCommandFile** method can be used to change the name of the local file or directory that is the target of the command, however using this method to redirect access to a resource can have unintended side-effects, particularly in the case where the URL provided by the client actually resolves to an executable CGI program that handles the request.

If the client has provided a URL that resolves to a CGI program that handles the request, this method behaves differently than if the URL is resolved to a local file or directory. If the client uses the GET or POST command that results in a program being executed to handle the request, this method will return the path to the server root directory along with any additional path information provided in the URL. In other words, the file name returned by this method will be the same as the PATH_TRANSLATED value passed to the CGI program.

This method should only be called after the client request has been received by the server, typically inside a **OnCommand** or **OnExecute** event handler. It should not be called inside a **OnConnect** event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualPath, GetCommandUrl, RedirectRequest, SetCommandFile, OnCommand, OnExecute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandLine Method

```
INT GetCommandLine(
   UINT nClientId,
   LPTSTR lpszCmdLine,
   INT nMaxLength
);

INT GetCommandLine(
   UINT nClientId,
   CString& strCmdLine
);
```

Return the complete command line issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A null-terminated string buffer that will contain the command, including all arguments. The string buffer will be null-terminated and must be large enough to store the complete command line. If this parameter is NULL, the method will return the length of the command line. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **GetCommandLine** method is used to obtain the command that was issued by the client, and is commonly used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. When the method returns, the string buffer provided by the caller will contain the complete command, including the resource path and requested HTTP version. Any extraneous whitespace will be removed, however all encoding will be preserved.

To obtain the complete URL associated with the request issued by the client, use the **GetCommandUrl** method. If the command sent by the client is used to perform an action on a file or directory, the **GetCommandFile** method should be called to obtain the full path to the local file rather than using the resource path.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandUrl, OnCommand, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandName Method

```
INT GetCommandName
   UINT nClientId,
   LPTSTR lpszCommand,
   INT nMaxLength
);

INT GetCommandName
   UINT nClientId,
   CString& strCommand
);
```

Return the name of the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string buffer that will contain the command name. The string buffer will be null-terminated and must be large enough to store the complete parameter value. If this parameter is NULL, the method will only return the length of the current command in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszCommand* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **GetCommandName** method is used to obtain the name of the last command that was issued by the client. The command name returned by this method will always be capitalized, regardless of how it was sent by the client. This method is typically used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. It should not be called inside a **OnConnect** event handler because the server has not processed the client request at that point.

The **GetCommandUrl** function can be used to return the resource that was requested by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandUrl, OnCommand, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandQuery Method

```
INT GetCommandQuery(
   UINT nClientId,
   LPTSTR lpszParameters,
   INT nMaxLength
);

INT GetCommandQuery(
   UINT nClientId,
   CString& strParameters
);
```

Return the query parameters included with the command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszParameters

A null-terminated string buffer that will contain the query parameters when the method returns. The string buffer will be null terminated up to the maximum number of characters specified by the caller. If this parameter is NULL the method will return the length of the query string. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszParameters* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the client did not provide any query parameters, this method will return zero and the last error code will be zero.

Remarks

The **GetCommandQuery** method is used to obtain a copy of the query parameters that were included in the request URL. If there were no query parameters, the string buffer will be empty and the return value will be zero. If the request did include query parameters, they will be returned to the caller in their original, encoded form.

This method should not be called within an **OnConnect** event handler because the client request has not been processed at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandName, GetCommandResource, ReceiveRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandResource Method

```
INT GetCommandResource(
   UINT nClientId,
   LPTSTR lpszResource,
   INT nMaxLength
);

INT GetCommandResource(
   UINT nClientId,
   CString& strUrlPath
);
```

Return the URL path for the resource requested by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResource

A null-terminated string buffer that will contain the URL path provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete path. If this parameter is NULL, the method will only return the length of the path. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the ***lpszResource*** parameter is NULL, this value should be zero. If this value is less than the length of the URL, the method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetCommandResource** method returns the URL path that the client provided to access the requested resource. The path will not include the URI scheme, user credentials or query parameters. The **GetCommandFile** method can be used to determine the name of the local file on the server that will be accessed using this URL.

If you require the complete URL, not just the path to the resource, use the **GetCommandUrl** method.

This method should only be called after the client request has been received by the server, typically inside an **OnCommand** event handler. It should not be called inside a **OnConnect** event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandQuery, GetCommandUrl, RedirectRequest

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandResult Method

```
INT GetCommandResult(
   UINT nClientId,
   LPTSTR lpszResult,
   INT nMaxLength
);

INT GetCommandResult(
   UINT nClientId,
   CString& strResult
);
```

Return the result code and description for the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A null-terminated string buffer that will contain the description of the result code. The string buffer will be null-terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **GetCommandResult** method is used to determine the result of the last command that was issued by the client and is typically called in the **OnResult** event handler. This method should only be called after a command has been processed or the **SendResponse** method has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code and a brief description of the result. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range are informational and only used with version 1.1 of the protocol. If the server returns a result code in this range, it means that it has received the request and that the client should proceed.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. In most cases this means that the server has returned the requested data to the client, however if a 204 result code is sent, this indicates that the request has been processed but there is no data available.
300-399	Result codes in this range indicate that the requested resource has been moved to a

	new location. The most common result codes are 301 and 302. A value of 301 indicates that the location of the resource has changed permanently and all future requests should be sent to the new URL. A value of 302 indicates that the resource location has changed temporarily. The new location of the resource is sent to the client by setting the Location response header field.
400-499	Result codes in this range indicate an error with the request that was made by the client. These types of errors include invalid commands, requests that can only be issued by authenticated clients, or resources that cannot be accessed on the server. The most common result code in this range is the 404 code which indicates that the requested document could not be found.
500-599	Result codes in this range indicate a server error has occurred while processing a valid request. These types of errors are returned when a command has not been implemented, the execution of a CGI program or script has failed unexpectedly, or an internal server error has occurred.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ReceiveRequest, SendResponse, OnResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandUrl Method

```
INT GetCommandUrl(
   UINT nClientId,
   LPTSTR lpszUrl,
   INT nMaxLength
);

INT GetCommandUrl(
   UINT nClientId,
   CString& strUrl
);
```

Return the complete URL of the resource requested by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUrl

A null-terminated string buffer that will contain the URL provided by the client for the current command. The string buffer will be null-terminated and must be large enough to store the complete URL. If this parameter is NULL, the method will only return the length of the URL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the ***lpszUrl*** parameter is NULL, this value should be zero. If this value is less than the length of the URL, the method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetCommandUrl** method returns the complete URL that the client provided to access the requested resource. The URL will include any query parameters that were specified by the client, but it will not include any user credentials. The **GetCommandFile** method can be used to determine the name of the local file on the server that will be accessed using this URL.

If you only require the URL path, without the URI scheme or the query parameters, use the **GetCommandResource** method.

This method should only be called after the client request has been received by the server, typically inside an **OnCommand** event handler. It should not be called inside an **OnConnect** event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandFile, GetCommandResource, RedirectRequest, OnCommand

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetDirectory Method

```
INT GetDirectory(
   LPTSTR lpszDirectory,
   INT nMaxLength
);

INT GetDirectory(
   CString& strDirectory
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

lpszDirectory

A null-terminated string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetDirectory** method will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, the current working directory for the process will be used and this method can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This method will return the fully resolved path name, with all environment variables expanded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetIdentity, GetName, SetDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetHandle Method

```
HSERVER GetHandle();
```

The **GetHandle** method returns the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If the server is inactive, the value INVALID_SERVER will be returned.

Remarks

This method is used to obtain the server handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetIdentity Method

```
INT GetIdentity(
   LPTSTR lpszIdentity,
   INT nMaxLength
);

INT GetIdentity(
   CString& strIdentity
);
```

Return the identity of the specified server.

Parameters

lpszIdentity

A null-terminated string buffer that will contain the identity of the server when the method returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetIdentity** method returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number. The **SetIdentity** method can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetLastError Method

```
DWORD GetLastError(
   LPTSTR lpszError,
   INT nMaxLength
);

DWORD GetLastError(
   CString& strError
);

DWORD GetLastError();
```

Return the last server error code and a description of the error.

Parameters

lpszError

A null-terminated string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the ***lpszError*** parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods clear the last error code when they succeed.

It is important to note that the error codes returned by this method are different than the command result codes that are defined in RFC 2616, the standard protocol specification for HTTP. This method is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **SendResponse** method is used to send responses to the client, and the **GetCommandResult** method can be used to determine the result of the last command sent by the client.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR. Those methods which clear the last error code when they succeed are noted on their reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandResult, SendResponse, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetLogFile Method

```
BOOL GetLogFile(
   UINT *lpnLogFormat,
   UINT *lpnLogLevel,
   LPTSTR lpszFileName,
   INT nMaxLength
);

BOOL GetLogFile(
   UINT *lpnLogFormat,
   UINT *lpnLogLevel,
   CString& strFileName
);

BOOL GetLogFile(
   CString& strFileName
);
```

Return the current log file format and the full path to the file.

Parameters

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the method returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Value	Description
HTTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON	This value specifies that the server should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A null-terminated string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required. An alternate version of this method accepts a **CString**

object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the ***lpszFileName*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **HTTPSERVERCONFIG** structure definition for a list of supported log file formats. If logging has not been enabled, this method will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This method can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **RenameServerLogFile** method to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **SetLogFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RenameServerLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero and specifies the amount of memory allocated by the server. If the server is inactive or cannot be locked, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the server into a locked state, and all client sessions will block until the method returns. Because this method enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this method can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetStackSize, SetStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetName Method

```
INT GetName(
   UINT nClientId,
   LPTSTR lpszHostName,
   INT nMaxLength
);

INT GetName(
   UINT nClientId,
   CString& strHostName
);
```

Return the host name assigned to the specified server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will return the host name assigned to the specified server. If the ***nClientId*** parameter has a value of zero, the method will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the ***nClientId*** parameter specifies a client session, then it this method will return the host name that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetOptions Method

```
DWORD GetOptions();
```

Return the options specified for this instance of the server.

Parameters

None.

Return Value

The current server options. For a list of available options, see Server Option Constants

Remarks

The **GetOptions** method returns the default options for the current instance of the server. To change the server options, use the **SetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetPriority Method

```
INT GetPriority();
```

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Value	Description
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: cshtsv12.lib

See Also

SetPriority, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetProgramExitCode Method

```
BOOL GetProgramExitCode(
   UINT nClientId,
   DWORD& dwExitCode
);
```

Return the exit code of the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwExitCode

An unsigned integer that will contain the program exit code when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetProgramExitCode** method returns the exit code of a registered CGI program or script that was executed. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by programs to communicate status information back to the server application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetProgramOutput, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetProgramName Method

```
INT GetProgramName(
   UINT nClientId,
   LPTSTR lpszProgramName,
   INT nMaxLength
);

INT GetProgramName(
   UINT nClientId,
   CString& strProgramName
);
```

Return the name of the CGI program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A null-terminated string buffer that will contain the name of the CGI program executed by the client. This parameter cannot be NULL and should be at least MAX_PATH characters in size. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the method succeeds, the return value is the length of the executable file name. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed a CGI program this method will return zero and the last error code will be set to zero.

Remarks

The **GetProgramName** method returns the local file name of the CGI program that was executed in response to a request from the client. This is the full path to the executable that was registered with the server using the **RegisterProgram** method. If the client specified a regular document or directory, this method will return a value of zero, indicating that no CGI program has been executed to handle the request.

To obtain the resource URL that was provided by the client, use the **GetCommandUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramOutput, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetProgramOutput Method

```
DWORD GetProgramOutput(
   UINT nClientId,
   LPBYTE lpBuffer,
   DWORD dwBufferSize
);
```

Return a copy of the standard output from the a CGI program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the method will return the number of bytes of data that was output by the program. Note that this output is not null-terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the ***lpBuffer*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of bytes copied into the specified buffer. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramOutput** method is used to obtain a copy of the output generated by a CGI program. To determine the number of bytes of output available to read, call this method with the ***lpBuffer*** parameter as NULL and the ***dwBufferSize*** parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null-terminated.

This method returns the raw output from the program which may contain a response header block, escape sequences, control characters and embedded nulls. When the application processes the output returned by this method, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the program as a null-terminated string, use the **GetProgramText** method.

This method should only be used within an **OnExecute** event handler, which occurs after the program has terminated.

Example

```
LPBYTE lpBuffer = NULL;  // The output buffer
DWORD cbBuffer = 0;      // Number of bytes in the output buffer

// Determine the number of bytes in the output buffer
cbBuffer = pHttpServer->GetProgramOutput(nClientId, NULL, 0);

if (cbBuffer > 0)
{
```

```
    // Allocate memory for the buffer
    lpBuffer = new BYTE[cbBuffer + 1];

    // Copy the program output to the buffer
    cbBuffer = pHttpServer->GetProgramOutput(nClientId, lpBuffer, cbBuffer + 1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    delete lpBuffer;
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetProgramExitCode, GetProgramText, OnExecute, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetProgramText Method

```
INT GetProgramText(
   UINT nClientId,
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT GetProgramText(
   UINT nClientId,
   CString& strBuffer
);
```

Return a copy of the standard output from a CGI program in a string buffer.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the method will return the number of bytes of characters that was output by the program, not including a terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the ***lpszBuffer*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramText** method is used to obtain a copy of the output generated by a CGI program. To determine the number of characters of output available to read, call this method with the ***lpszBuffer*** parameter as NULL and the ***nMaxLength*** parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This method will only return textual output from the program and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the program, use the **GetProgramOutput** method. If the program outputs a response header block, this will be included in the string buffer.

This method should only be used within an **OnExecute** event handler, which occurs after the program has terminated.

Example

```
CString strBuffer;
```

```
if (pHttpServer->GetProgramText(nClientId, strBuffer) > 0)
    pEditCtrl->SetWindowText(strBuffer);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramOutput, OnExecute, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetStackSize Method

```
DWORD GetStackSize();
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

If this method is called when the server is not active, it will return the default stack size that the server will be configured to use when it starts. In this case, a return value of zero specifies that the default stack size will be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: cshtsv12.lib

See Also

GetMemoryUsage, SetStackSize, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetTransferInfo Method

```
BOOL GetTransferInfo(
   UINT nClientId,
   LPHTTPSERVERTRANSFER lpTransferInfo
);
```

Return information about the current file transfer.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **HTTPSERVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the ***dwSize*** member of the structure must be initialized to specify the structure size prior to calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method should only be called after the client has issued the GET or PUT commands to initiate a file transfer, otherwise the return value will be zero.

Remarks

The **GetTransferInfo** method is used to obtain information about the last file transfer that was performed by the client. This method is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ReceveRequest, SendResponse, HTTPSERVERTRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetUuid Method

```
INT GetUuid(
   LPTSTR lpszHostUuid,
   INT nMaxLength
);

INT GetUuidString(
   CString& strHostUuid
);
```

Return the UUID assigned to the server as a printable string.

Parameters

lpszHostUuid

A null-terminated string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete UUID string, the method will return a value of zero.

Remarks

The **GetUuid** method returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetUuid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetVirtualHostId Method

```
UINT GetVirtualHostId(
   LPCTSTR lpszHostName,
   UINT nHostPort
);
```

Return the virtual host ID associated with the specified hostname.

Parameters

lpszHostName

A string that specifies the virtual host name.

nHostPort

An optional integer value which specifies the port number for the virtual host. If this parameter is specified it must be the same value as the original port number that the server was configured to use.

Return Value

If the method succeeds, the return value is the host ID that uniquely identifies the virtual host. If the server handle is invalid, or there is no virtual host with the specified name, the method will return VIRTUAL_HOST_UNKNOWN. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **GetVirtualHostId** method is used to obtain the unique virtual host ID that is associated with a specific hostname. This method will match both the primary virtual hostname added using the **AddVirtualHost** method, as well as any aliases that were added using the **AddVirtualHostAlias** method. To obtain the virtual host ID associated with the active client session, use the **GetClientVirtualHostId** method.

The *nHostPort* parameter should either be omitted or specified with the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, AddVirtualHostAlias, DeleteVirtualHost, GetClientVirtualHostId, GetVirtualHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetVirtualHostName Method

```
INT GetVirtualHostName(
   UINT nHostId,
   LPTSTR lpszHostName,
   INT nMaxLength
);

INT GetVirtualHostName(
   UINT nHostId,
   CString& strHostName
);
```

Return the hostname associated with the specified virtual host ID.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszHostName

A null-terminated string buffer that will contain the virtual host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. If this parameter is NULL, the method will return the length of the virtual hostname. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. If the ***lpszHostName*** parameter is NULL this value must be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the host ID is invalid, or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

The **GetVirtualHostName** method returns the primary hostname associated with the specified virtual host ID. This is the same hostname that was specified when the virtual host was added to the server configuration using the **AddVirtualHost** method. To obtain the hostname that was used by the active client session to connect to the server, use the **GetClientVirtualHost** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualHost, AddVirtualHostAlias, DeleteVirtualHost, GetClientVirtualHost, GetVirtualHostId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::IsActive Method

```
BOOL IsActive();
```

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

CHttpServer, IsListening, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::IsClientAuthenticated Method

```
BOOL IsClientAuthenticated(
    UINT nClientId
);
```

Determine if the specified client session has been authenticated.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this method will return a non-zero value, otherwise it will return zero. If the client ID is valid, this method will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

AuthenticateClient, GetClientCredentials, RequireAuthentication

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::IsCommandEnabled Method

```
BOOL IsCommandEnabled(
   LPCTSTR lpszCommand
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

lpszCommand

A null-terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the command is enabled, this method will return a non-zero value. If the command is disabled or the command name does not match a supported command, this method will return zero.

Remarks

The **IsCommandEnabled** method is used to determine whether a specific command is enabled. Typically this method is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled using the **EnableCommand** method and disabled using the **DisableCommand** method.

This method does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **AuthenticateClient** method, and certain commands can be limited by the permissions granted to the client. For example, even if the PUT command is enabled, a client must have the HTTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see User Access Constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AuthenticateClient, DisableCommand, EnableCommand, GetCommandName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

CHttpServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::IsListening Method

```
BOOL IsListening();
```

Determine if the server is listening for client connections.

Return Value

This method returns a non-zero value if the server has been started and is listening for client connections. If the server is stopped or has been suspended this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

CHttpServer, IsActive, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(
   HSERVER hServer,
   UINT nClientId,
   UINT nEventId,
   DWORD dwError,
   BOOL& bHandled
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see Server Event Constants.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::ReceiveRequest Method

```
BOOL ReceiveRequest(
   UINT nClientId,
   DWORD dwOptions,
   LPHTTPREQUEST lpRequest,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength
);
```

Receive the request that was sent by the client to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be copied. It may be one of the following values:

Value	Description
HTTP_REQUEST_DEFAULT	If the ***lpdwLength*** parameter points to a integer with a non-zero value, the ***lpvBuffer*** parameter is considered to be a pointer to a block of memory that has been allocated to store the request data. This is the same as specifying the HTTP_REQUEST_MEMORY option. If the ***lpdwLength*** parameter points to an integer with a value of zero, the ***lpvBuffer*** parameter is considered to be a pointer to an HGLOBAL memory handle. This is the same as specifying the HTTP_REQUEST_HGLOBAL option.
HTTP_REQUEST_MEMORY	The ***lpvBuffer*** parameter is a pointer to a block of memory that has been allocated to store the request data. The maximum number of bytes of data that can stored is determined by the value of the integer that the ***lpdwLength*** parameter points to. When the method returns, that value will updated with with actual number of bytes copied into the buffer.
HTTP_REQUEST_STRING	The ***lpvBuffer*** parameter is a pointer to a string buffer that has been allocated to store the request data. The maximum number of bytes of data that can stored is determined by the value of the integer that the ***lpdwLength*** parameter points to. When the method returns, that value will updated with with actual number of bytes copied into the buffer.
HTTP_REQUEST_HGLOBAL	The ***lpvBuffer*** parameter is a pointer to an HGLOBAL memory handle. When the method returns, the handle will reference a block of memory that contains the request data submitted by the client. The ***lpdwLength*** parameter will contain the number of bytes copied to the buffer.
HTTP_REQUEST_FILE	The ***lpvBuffer*** parameter is a pointer to a string which

	specifies the name of a file that will contain the request data. If the file does not exist, it will be created. If it does exist, the contents will be replaced. This option is typically used in conjunction with the PUT command. If the ***lpdwLength*** parameter is not NULL, the value it points to will be updated with the actual number of bytes stored in the file.
HTTP_REQUEST_HANDLE	The ***lpvBuffer*** parameter is a handle to an open file. This option is typically used in conjunction with the POST or PUT commands. If the ***lpdwLength*** parameter is not NULL, the value it points to will be updated with the actual number of bytes written to the file. If this option is specified, the request data will be written from the current position in the file and will advance the file pointer by the number of bytes received from the client.

lpRequest

A pointer to a **HTTPREQUEST** structure which contains information about the request from the client. This parameter cannot be NULL. The structure that is passed to this method must have all members set to a value of zero except the ***dwSize*** member, which must be initialized to the size of the structure.

lpvBuffer

The buffer that will contain any request data that was submitted by the client. The ***dwOptions*** parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. If this parameter is NULL, any data submitted by the client will not be copied.

lpdwLength

A pointer to an unsigned integer that will contain the number of bytes of data submitted by the client when the method returns. If the ***lpvBuffer*** parameter specifies a memory or string buffer, this value must be initialized to the maximum size of the buffer before the method is called. If the ***lpvBuffer*** parameter points to a global memory handle, this value must be initialized to zero. If ***lpvBuffer*** is NULL or specifies a file name, this parameter may be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ReceiveRequest** method is called within an **OnCommand** event handler to process the command issued by the client and return information about the request to the server application. It is only necessary for the application to call this method if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

This method may only be called once per command issued by the client and the data referenced in the **HTTPREQUEST** structure only remains valid while the client is connected to the server. You should never attempt to directly modify the data referenced by any of the structure members. If you wish to store or modify any of the string values returned in the structure, you should allocate a buffer large enough to store the contents of the string, including the terminating null character, and copy the

string into that buffer.

If the HTTP_REQUEST_HGLOBAL option is used to return a copy of the request data in a global memory buffer, the HGLOBAL handle must be freed by the application when the data is no longer needed. Failure to free this handle will result in a memory leak.

If the HTTP_REQUEST_HANDLE option is used to write a copy of the request data to an open file, the handle must reference a disk file that was opened or created using the **CreateFile** function with GENERIC_WRITE access. It cannot be a handle to a device or named pipe. If the method succeeds, the file pointer is advanced by the number of bytes of request data submitted by the client. If the method fails, the file pointer is returned to its original position prior to the method being called.

Example

```
// Initialize the HTTPREQUEST structure
HTTPREQUEST httpRequest;
ZeroMemory(&httpRequest, sizeof(httpRequest));
httpRequest.dwSize = sizeof(httpRequest);

// Return the data in a global memory buffer
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;

bSuccess = pServer->ReceiveRequest(nClientId,
                                   HTTP_REQUEST_HGLOBAL,
                                   &httpRequest,
                                   &hgblBuffer,
                                   &dwLength);

if (bSuccess && hgblBuffer != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    if (lpBuffer != NULL)
    {
        // Process dwLength bytes of data submitted by the client
    }

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SendResponse, HTTPREQUEST

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::RedirectRequest Method

```
BOOL RedirectRequest(
   UINT nClientId,
   UINT nMethod,
   LPCTSTR lpszLocation
);
```

Redirect the request from the client to another URL.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nMethod

An integer value that specifies if the redirection is permanent or temporary. The following values may be used:

Value	Description
HTTP_REDIRECT_PERMANENT	This value is used for permanent redirection, indicating that the client should update any record of the link with the new URL specified by the ***lpszLocation*** parameter. This result is cacheable and when the client makes subsequent requests for the resource, it should always use the new URL.
HTTP_REDIRECT_TEMPORARY	This value is used for temporary redirection, indicating that the client should issue a request for the resource using the new URL specified by the ***lpszLocation*** parameter, but subsequent requests should continue to use the original URL.
HTTP_REDIRECT_OTHER	This value is used for temporary redirection, however it instructs the client that it should use the GET command to request the redirected resource. This option is typically used to redirect a client after it has used the POST command.

lpszLocation

A null-terminated string that specifies the new location for the requested resource. This value must be a complete URL, including the http:// or https:// scheme. This parameter cannot be NULL or point to zero-length string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RedirectRequest** method can be used within a HTTP_CLIENT_COMMAND event handler to redirect the client to a new location for the resource that it has requested. This redirection can be permanent or temporary, depending on whether the server expects the client to continue to use the original URL when requesting the resource.

If the HTTP_REDIRECT_TEMPORARY method is used, the actual status code that is returned to the client depends on the version of the protocol that is being used. If the client has issued the request using HTTP 1.0 then the server will return a 302 code to the client. If the client is using HTTP 1.1, the server will return a 307 code to the client that indicates it should use the same command verb (GET, POST, etc.) when requesting the resource at the new location.

If the HTTP_REDIRECT_OTHER method is used, the status code that is returned to the client depends on which version of the protocol is being used. For clients who are using HTTP 1.0, the server will return a 302 code to the client just as with the HTTP_REDIRECT_TEMPORARY method. If the client is using HTTP 1.1, the server will return a 303 code to the client that indicates it should always use the GET command to request the new resource, regardless if a different command was originally used (POST, PUT, etc.)

This method provides a simplified interface for sending a redirection status code that also implicitly sets the Location response header to the value of the ***lpszLocation*** parameter. If the server application needs to send alternate redirection codes such as 305 (Use Proxy) then it should use the **SendReponse** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RequireAuthentication, SendErrorResponse, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::RegisterHandler Method

```
BOOL RegisterHandler(
   UINT nHostId,
   LPCTSTR lpszExtension,
   LPCTSTR lpszProgramFile,
   LPCTSTR lpszParameters,
   LPCTSTR lpszDirectory
);
```

Register a CGI program for use and associate it with a file name extension.

Parameters

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszExtension

A null-terminated string which specifies the file name extension that is associated with the CGI program. This parameter cannot be NULL.

lpszProgramFile

A null-terminated string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A null-terminated string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A null-terminated string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RegisterHandler** method registers an executable CGI program and associates it with a file name extension. When the client issues a GET or POST command that specifies a file with that extension, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple file name extensions that reference the same program. The only requirement is that the extension be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a

script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the full path to the local script filename. If no placeholder is included in the parameters, or *lpszParameters* is a NULL pointer, the script file name will be passed to the program as its only argument.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **GetProgramOutput** method from within an **OnExecute** event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
// Register a handler for VBScript
pServer->RegisterHandler(hServer,
                         VIRTUAL_HOST_DEFAULT,
                         _T("vbs"),
                         _T("%SystemRoot%\\System32\\cscript.exe"),
                         _T("/nologo /b \"%1\""),
                         NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramOutput, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::RegisterProgram Method

```
BOOL RegisterProgram(
   LPCTSTR lpszVirtualPath,
   LPCTSTR lpszProgramFile,
);

BOOL RegisterProgram(
   LPCTSTR lpszVirtualPath,
   LPCTSTR lpszProgramFile,
   LPCTSTR lpszParameters,
   LPCTSTR lpszDirectory
);
```

Register a CGI program for use and associate it with a virtual path on the server.

Parameters

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszVirtualPath

A null-terminated string which specifies the virtual path to the CGI program. This must be an absolute path, but does not have to specify a pre-existing virtual path or map to the directory structure of the root document directory for the server. This parameter cannot be NULL. The maximum length of the virtual path is 1024 characters.

lpszProgramFile

A null-terminated string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A null-terminated string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A null-terminated string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **RegisterProgram** function registers an executable CGI program and associates it with a virtual path. When the client issues a GET or POST command specifying the virtual path associated with the program, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple virtual paths that

reference the same executable file. The only requirement is that the virtual path be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by ***lpszProgramFile*** must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The ***lpszParameters*** string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the virtual path associated with the program. If ***lpszParameters*** is NULL or a zero-length string, then no additional parameters are passed to the program.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **GetProgramOutput** function from within an **OnExecute** event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
m_pServer->RegisterProgram(hServer,
                           VIRTUAL_HOST_DEFAULT,
                           HTTP_METHOD_DEFAULT,
                           _T("/order/invoice"),
                           _T("%ProgramData%\\MyServer\\Programs\\invoice.exe"),
                           NULL,
                           NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramExitCode, GetProgramOutput, RegisterHandler

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::RenameServerLogFile Method

```
BOOL RenameServerLogFile(
   LPCTSTR lpszFileName
);
```

Rename or delete the current log file being updated by the server.

Parameters

lpszFileName

A null-terminated string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If logging is not currently enabled for the server, this method will return zero.

Remarks

The **RenameServerLogFile** method is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the *lpszFileName* parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This method must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **GetLogFile** method to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** Windows API functions with that file.

To disable logging, use the **SetLogFile** method and specify the logging format as HTTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLogFile, SetLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::RequireAuthentication Method

```
BOOL RequireAuthentication(
   UINT nClientId,
   UINT nAuthType,
   LPCTSTR lpszRealm
);
```

Send a response to the client indicating that authentication is required.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nAuthType

An integer value that corresponds to a result code, informing the client if the redirection is permanent or temporary. The following values may be used:

Value	Description
HTTP_AUTH_BASIC	This option specifies the Basic authentication scheme should be used. This option is supported by all clients that support at least version 1.0 of the protocol.

lpszRealm

A null-terminated string that is displayed by a web browser to indicate to the user which username and password they should use. If this parameter is NULL or an empty string, the domain name the client used to establish the connection will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RequireAuthentication** method can be used within an **OnCommand** event handler to indicate to the client that it must provide a username and password to access the requested resource. The client should respond by issuing another request that includes the required credentials. To determine if a client has included credentials with its request, use the **IsClientAuthenticated** method. The **GetClientCredentials** method will return the username and password that was provided by the client.

Some clients may require that the session be secure if authentication is requested or display warning messages to the user if the connection is not secure. It is recommended that you enable security using the HTTP_OPTION_SECURE option if your application will require clients to authenticate before accessing specific resources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientCredentials, IsClientAuthenticated, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::Restart Method

```
BOOL Restart();
```

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::Resume Method

```
BOOL Resume();
```

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

Restart, Start, Stop, Suspend, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SendErrorResponse Method

```
BOOL SendErrorResponse(
   UINT nClientId,
   UINT nErrorCode,
   LPCTSTR lpszMessage
);
```

Send a customized error response to the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nErrorCode

An integer value that specifies the error code that should be sent to the client. This value should correspond to the error result codes defined for HTTP in RFC 2616, which are three-digit values in the range of 400 through 599. The method will fail if an invalid error code is specified.

lpszMessage

A null-terminated string that describes the error. If this parameter is NULL or specifies a zero-length string, a default message will be selected based on the error code.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SendErrorResponse** method sends a response to the client indicating that an error has occurred, providing a numeric error code and HTML formatted text which may be displayed to the user. The *lpszMessage* parameter should provide a brief description of the error that will be included in the output sent to the client. Note that the message should not contain any special formatting control characters or HTML markup.

This method provides a simplified interface for sending an error response to the client. In some cases, a browser may choose to display its own error message to the user in place of the generic HTML document generated by this method. If you want your application to send a customized HTML document for a specific type of error, you should use the **SendResponse** method.

If you wish to redirect the client to use an alternate URL to access the requested resource, it is recommended that you use the **RedirectRequest** method rather than sending an error response.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RedirectRequest, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SendResponse Method

```
BOOL SendResponse(
   UINT nClientId,
   DWORD dwOptions,
   LPHTTPRESPONSE lpResponse,
   LPVOID lpvBuffer,
   DWORD dwLength
);
```

Send a response from the server to the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be copied. It may be one of the following values:

Value	Description
HTTP_RESPONSE_DEFAULT	The ***lpvBuffer*** parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the ***dwLength*** parameter. This is the same as specifying the HTTP_RESPONSE_MEMORY option.
HTTP_RESPONSE_MEMORY	The ***lpvBuffer*** parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the ***dwLength*** parameter. The data will be sent to the client as a stream of bytes. If the server application was compiled using Unicode, it is responsibility of the application to convert any Unicode text to either ANSI or UTF-8, depending on the resource that was requested by the client.
HTTP_RESPONSE_STRING	The ***lpvBuffer*** parameter is a pointer to a string buffer that contains the response data. The maximum number of bytes of data that will be sent to the client is determined by the ***dwLength*** parameter. If the value of the ***dwLength*** parameter exceeds the string length, the value will be ignored and the contents of the string will be sent up to the terminating null character. If the Unicode version of this method is called, the string will be converted to a byte array before being sent to the client.
HTTP_RESPONSE_HGLOBAL	The ***lpvBuffer*** parameter is an HGLOBAL memory handle which references a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the ***dwLength*** parameter. The data will be sent to the client as a stream of bytes. It is the responsibility of the application to free the global memory

	handle after it is no longer needed.
HTTP_RESPONSE_FILE	The ***lpvBuffer*** parameter is a pointer to a string which specifies the name of a file that contains the response data. If the file does not exist, or does not specify a regular file, this method will fail. The ***dwLength*** parameter is ignored. If the content type for the specified file is not explicitly defined in the response, the method will attempt to automatically determine the correct type based on the file name extension and/or the contents of the file.
HTTP_RESPONSE_HANDLE	The ***lpvBuffer*** parameter is a handle to an open file and the ***dwLength*** parameter specifies the number of bytes to be read from the file and send to the client. If this option is specified, the response data will be read from the current position in the file and will advance the file pointer by the number of bytes sent to the client.
HTTP_RESPONSE_DYNAMIC	The response data will be generated dynamically. This prevents the content length from being included in the response header, and forces the connection to close, regardless if the keep-alive option has been specified. This option must be specified if the application wishes to use the **SendResponseData** method to send additional data to the client.
HTTP_RESPONSE_NOCACHE	Informs the client that the data being returned by the server should not be cached. Typically this is used in conjunction with the HTTP_RESPONSE_DYNAMIC option when the data is being generated dynamically.

lpResponse

A pointer to a **HTTPRESPONSE** structure which contains additional information about the response to the client. The structure that is passed by reference to this method must have the ***dwSize*** member initialized to the size of the structure or the method will fail. This parameter may be NULL, in which case a default response of "200 OK" is sent to the client along with any data specified by the ***lpvBuffer*** parameter.

lpvBuffer

The buffer that will contain any response data that should be sent to the client. The ***dwOptions*** parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. This parameter may be NULL, in which case no data will be sent to the client. If this method is called in response to a HEAD command being sent by the client, this parameter is ignored.

dwLength

An unsigned integer that specifies the number of bytes of data to be sent to the client. If the ***lpvBuffer*** parameter is NULL, this value must be zero. If this method is called in response to a HEAD command being sent by the client, this parameter is ignored.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SendResponse** method is called within an **OnCommand** event handler to respond to the request made by the client. This method may only be called once after a command has been received, and must be called after the **ReceiveRequest** method. It is only necessary for the application to call this method if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

If the HTTP_RESPONSE_HANDLE option is used to read a copy of the response data from an open file, the handle must reference a disk file that was opened using the **CreateFile** function with GENERIC_READ access. It cannot be a handle to a device or named pipe. If the ***dwLength*** parameter is larger than the total number of bytes available to be read from the current position in the file, the method will stop sending data to the client when it reaches the end-of-file. If the method succeeds, the file pointer is advanced by the number of bytes of response data sent to the client. If the method fails, the file pointer is returned to its original position prior to the method being called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RecieveRequest, SendResponseData, OnCommand, HTTPRESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SendResponseData Method

```
INT SendResponseData(
   UINT nClientId,
   LPBYTE lpBuffer,
   INT nLength
);
```

Send additional data to the client in response to a command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A buffer that contains the response data that should be sent to the client. This parameter cannot be NULL.

nLength

An integer that specifies the number of bytes of data in the buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes of data sent to the client. If the method fails, the return value is HTTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **SendResponseData** method is called within a **OnCommand** event handler to send data to the client in response to a request. This method can only be used to send dynamically generated content after the **SendResponse** method has been called. The HTTP_RESPONSE_DYNAMIC option must have been specified when responding to the client or this method will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RecieveRequest, SendResponse, OnCommand, HTTPRESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetCertificate Method

```
BOOL SetCertificate(
   DWORD dwProtocol,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);

BOOL SetCertificate(
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);
```

Set the name of the certificate to be used with secure connections.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpszPassword

An optional pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only required if the ***lpszCertStore*** parameter specifies a certificate file in PKCS #12 format that is password protected.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call the **GetLastError** method.

Remarks

An application only needs to call this method to support secure connections using TLS. When a certificate is specified, it must have a private key associated with it. Attempting to use a certificate which does not have a private key will result in an error during the startup process indicating that the server credentials are invalid.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the class will be unable to access the certificate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetOptions, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetClientAccess Method

```
BOOL SetClientAccess(
   UINT nClientId,
   DWORD dwUserAccess
);
```

Change the access rights associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User and File Access Constants.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **SetClientAccess** method can change multiple access rights for the client session. The **EnableClientAccess** method can be used to grant or revoke a specific permission for the client session.

If the *dwUserAccess* parameter has a value of HTTP_ACCESS_DEFAULT, then default permissions will be granted to the client session based on the configuration of the server. This is the recommended value for most clients. It is important to consider the implications of changing the access permissions granted to a client session. For example, if you do not grant clients HTTP_ACCESS_READ permission, it can effectively disable the site because the server will return 403 Forbidden errors for all GET and HEAD requests.

This function should typically be called in the **OnConnect** event handler to assign general permissions to the client, or in the **OnCommand** event handler after the client has issued a request and provided any authentication credentials that are required.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

AuthenticateClient, EnableClientAccess, GetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetClientHeader Method

```
BOOL HttpSetClientHeader(
   UINT nClientId,
   UINT nHeaderType,
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue
);
```

Create or change the value of a request or response header for the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to create or modify. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Change a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Change a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A null-terminated string that specifies the name of the header that should be created or modified. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A null-terminated string that specifies the new value of the header. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the header value from the list of request or response headers.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **SetClientHeader** method will change the value of a request or response header for the specified client session. If the *lpszHeaderName* value matches an existing header field, its value will be replaced. If the header name is not defined, then a new header will be created with the given value. You should not change the value of most standard response header values unless you are certain of the impact that it would have on the normal operation of the client.

If you wish to define a custom header value that would be included in the response to a client request,

you should prefix the header name with "X-" to avoid potential conflicts with the standard response headers. For example, if you wanted to identify a customer, you could create a header field with the name "X-Customer-ID" and set the value to the customer ID number. The client application would receive this custom header value as part of the response to its request for a document.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteClientHeader, GetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetClientIdleTime Method

```
UINT SetClientIdleTime(
   UINT nClientId,
   UINT nTimeout
);
```

Change the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the method succeeds, the return value is the previous client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetClientIdleTime** method is will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an **OnTimeout** event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **GetClientIdleTime** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetClientVariable Method

```
BOOL HttpSetClientVariable(
   UINT nClientId,
   LPCTSTR lpszName,
   LPCTSTR lpszValue
);
```

Create or change the value of a CGI environment variable for the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A null-terminated string that specifies the name of the environment variable that should be created or modified. Environment variables are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A null-terminated string that specifies the new value of the environment variable. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the variable from the environment block for the client session.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **SetClientVariable** method will change the value of a environment variable for the specified client session. If the *lpszName* value matches an existing variable, its value will be replaced. If the variable is not defined, then a new variable will be created with the given value. The value of an environment variable can be obtained using the **GetClientVariable** method.

The server will automatically create a number of different environment variables that will be passed to a program or script executed by the server. These variables are defined in RFC 3875 as part of the Common Gateway Interface (CGI) 1.1 specification. The following variables are defined by the server and should not be modified directly by the application:

Variable Name	Description
AUTH_TYPE	The authorization scheme used by the server to authenticate the client session
CONTENT_LENGTH	The length of the request data provided by the client
CONTENT_TYPE	The MIME type that identifies the type of content provided by the client
DOCUMENT_ROOT	The full path to the local document root directory on the server
GATEWAY_INTERFACE	The version of the Common Gateway Interface that is being used by the server

PATH_INFO	The resource or sub-resource that is to be returned by the program or script
PATH_TRANSLATED	The path information mapped to the server root document directory structure
QUERY_STRING	The URL encoded query parameters passed to the program or script
REMOTE_ADDR	The network address of the client sending the request to the server
REMOTE_HOST	The same value as the REMOTE_ADDR variable
REMOTE_USER	The username specified as part of the authentication credentials provided by the client
REQUEST_METHOD	The method used by the client to request the resource
REQUEST_URI	The URI for the script provided by the client
SCRIPT_FILENAME	The full path to the program or script on the server
SCRIPT_NAME	The path to the program or script specified by the client
SERVER_NAME	The hostname or IP address of the server that the client connected to
SERVER_PORT	The port number that the client used to connect to the server
SERVER_PORT_SECURE	This variable has a value of "1" if the client connection to the server is secure
SERVER_PROTOCOL	The version of the server protocol used
SERVER_SOFTWARE	The server identity string which specifies the application name and version

In addition to the environment variables listed, the server will also create variables that are prefixed with "HTTP_" that are set to the value of request headers that are not otherwise defined. For example, the HTTP_USER_AGENT variable will be set to the value of the User-Agent header provided by the client as part of the request.

Calling the **SetClientVariable** method within an **OnExecute** event handler will have no effect because it occurs after the CGI program or script has completed execution. To create or modify environment variables for the client session, it should be done within an **OnCommand** event handler.

This method will not change the environment block for the server process. Each client session is allocated its own private environment block which is inherited by the CGI program. When the client session terminates, the memory allocated for its environment is released.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientVariable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetCommandFile Method

```
BOOL SetCommandFile(
   UINT nClientId,
   LPCTSTR lpszFileName
);
```

Change the name of a file or directory that is the target of the current command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetCommandFile** method is used by the application to change the target file or directory name for the current command from within an **OnCommand** event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the GET command to download a file from the server, this method can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **GetCommandFile** method.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the root directory for the client session. Because the root document directory for the client can depend on the hostname that the client used when connecting to the server, it is recommended that you always use an absolute path. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this method to change the target file name.

Typically this method would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this method to provide access to data that is stored outside of the server root directory. Incorrect use of this method could expose the server to security risks or cause unpredictable behavior by client applications. In most cases it is preferable to use the **AddVirtualPath** method to create a virtual path or file name on the server, or the **RedirectRequest** method to request the client use a different URL to access the resource.

This method should only be called within the context of the **OnCommand** event, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this method will return zero and the last error code will be set to ST_ERROR_INVALID_COMMAND.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddVirtualPath, GetCommandFile, ReceiveRequest, RedirectRequest, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetDirectory Method

```
BOOL SetDirectory(
   LPCTSTR lpszDirectory
);
```

Specify the local directory that will be used as the server root directory.

Parameters

lpszDirectory

A null-terminated string that specifies the root directory for the server. If this parameter is NULL or a zero-length string, the server will use the current working directory as the root directory.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero.

Remarks

The **SetDirectory** method specifies the path to the local directory that should be used as the root document directory for the server.

You cannot change the server root directory after the server has started. To change the root directory, you must stop the server using the **Stop** method and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetIdentity Method

```
BOOL SetIdentity(
   LPCTSTR lpszIdentity
);
```

Change the identity of the specified server.

Parameters

lpszIdentity

A null-terminated string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero, otherwise it will return a value of zero.

Remarks

The **SetClientIdentity** method changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **GetIdentity** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetLastError Method

```
VOID SetLastError(
   DWORD dwError
);
```

Set the last error code for the specified server session.

Parameters

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR.

If the ***dwError*** parameter is specified with a value of zero, this effectively clears the error code for the last method that failed. Those methods which clear the last error code when they succeed are noted on their reference page.

Applications can retrieve the value saved by this method by calling the **GetLastError** method to determine the specific reason for failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetCommandResult, GetLastError, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetLogFile Method

```
BOOL SetLogFile(
   UINT nLogFormat,
   UINT nLogLevel,
   LPCTSTR lpszFileName
);
```

Change the current log format, level of detail and file name.

Parameters

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Value	Description
HTTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about GET, PUT and POST requests.
HTTP_LOGFILE_COMBINED	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session. This format typically generates the largest logfiles.

nLogLevel

An integer value that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of HTTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A null-terminated string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the method succeeds, the return value is non-zero. If one of the parameters are invalid, the method will return zero.

Remarks

The **SetLogFile** method can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current

log file contents when changing the format or ensure that a new log file is created. To do this, combine the ***nLogFormat*** parameter with the constant HTTP_LOGFILE_DELETE.

The higher the value of the ***nLogLevel*** parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record GET, POST and PUT requests, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = HTTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (pHttpServer->GetLogFile(&nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != HTTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = pHttpServer->SetLogFile(nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLogFile, RenameServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetName Method

```
BOOL SetName(
   UINT nClientId,
   LPCTSTR lpszHostName
);
```

Change the host name assigned to the specified server or client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the method succeeds, the return value is non-zero. If the client ID is invalid, or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will change the host name assigned to the specified client session. If the ***nClientId*** parameter has a value of zero, the method will change default host name that was assigned to the server as part of the server configuration. If the ***nClientId*** parameter specifies a valid client session and the ***lpszHostName*** parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **GetName** method will return the host name specified by the client, and **SetName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetOptions Method

```
BOOL SetOptions(
   DWORD dwOptions
);
```

Sets the default options for this instance of the server.

Parameters

dwOptions

An unsigned integer which specifies one or more options. For a list of available options, see Server Option Constants.

Return Value

If the method is successful, it will return a non-zero value, otherwise it will return a value of zero.

Remarks

The **SetOptions** method changes the default options for the current instance of the server. This method cannot be used to change the options for an active instance of the server. If the server is active, it must be stopped before calling this method. To get the current options, use the **GetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetPriority Method

```
INT SetPriority(
   INT nPriority
);
```

Change the priority assigned to the specified server.

Parameters

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Value	Description
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **SetPriority** method can be used to change the current priority assigned to the specified server. Client connections that are accepted after this method is called will inherit the new priority as their

default priority. Previously existing client connections will not be affected by this method.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: cshtsv12.lib

See Also

GetPriority, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetStackSize Method

```
BOOL SetStackSize(
   DWORD dwStackSize
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetMemoryUsage, GetStackSize, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::SetUuid Method

```
BOOL SetUuid(
   LPCTSTR lpszHostUuid
);
```

Assign a UUID to the current instance of the server.

Parameters

lpszHostUuid

A null-terminated string that specifies the server UUID, terminated with a null character. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This parameter may be NULL or point to an empty string, in which case a temporary UUID will be randomly generated for the server.

Return Value

If the method succeeds, the return value is non-zero, otherwise the method return a value of zero.

Remarks

The **SetUuid** method assigns a Universally Unique Identifier (UUID) to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. This method cannot be used to change the UUID after the server has been started. To determine the UUID assigned to an active server instance, use the **GetUuid** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetUuid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::Start Method

```
BOOL Start(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   DWORD dwOptions
);

BOOL Start(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort
);

BOOL Start(
   UINT nLocalPort
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to accept client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit TLS. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see Server Option Constants. If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

In most cases, the ***lpszLocalHost*** parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and

configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this method references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

Example

```
CHttpServer *pHttpServer = new CHttpServer();

// Initialize the server configuration
pHttpServer->SetName(_T("server.company.com"));
pHttpServer->SetUuid(_T("10000000-1000-1000-1000-100000000000"));
pHttpServer->SetDirectory(_T("%ProgramData%\\MyProgram\\Server"));
pHttpServer->SetLogFile(HTTP_LOGFILE_EXTENDED, 5,
_T("%ProgramData%\\MyProgram\\Server.log"));
pHttpServer->SetOptions(HTTP_SERVER_DEFAULT);

// Start the server
pHttpServer->Start(HTTP_PORT_DEFAULT);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumClients, Restart, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::Stop Method

```
BOOL Stop();
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

Restart, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::Suspend Method

```
BOOL Suspend();
```

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

Restart, Resume, Start, Stop, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::Throttle Method

```
BOOL Throttle(
   UINT nMaxClients,
   UINT nMaxClientsPerAddress,
   DWORD dwConnectionRate
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

Restart, Resume, Start, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer Event Handlers

Method	Description
OnAuthenticate	The client requested authentication
OnConnect	The client established a control connection to the server
OnCommand	The client issued a command to the server
OnDisconnect	The client has disconnected from the server
OnDownload	The client has downloaded a file from the server
OnError	The event handler encountered an error when processing a client event
OnExecute	The client has executed an external program on the server
OnResult	The command issued by the client has been processed by the server
OnTimeout	The client has exceeded the maximum allowed idle time
OnUpload	The client has uploaded a file to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnAuthenticate Method

```
virtual void OnAuthenticate(
   UINT nClientId,
   LPCTSTR lpszHostName,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

A virtual method that is invoked after the client has successfully downloaded a file from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A null-terminated string that specifies the host name that the client used to establish the connection.

lpszUserName

A null-terminated string that specifies the user name provided by the client.

lpszPassword

A null-terminated string that specifies the cleartext user password provided by the client.

Return Value

None.

Remarks

The **OnAuthenticate** event handler is invoked when the client has provided authentication credentials as part of the request for a document or other resource. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

The event handler can call the **AuthenticateClient** method to authenticate the client session. If the event handler does not authenticate the client, the server will perform its default authentication. To reject the credentials provided by the client, use the **SendErrorResponse** method with a result code of 401.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OnCommand, OnConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnCommand Method

```
virtual void OnCommand(
   UINT nClientId,
   LPCTSTR lpszCommand,
   LPCTSTR lpszResource,
   LPCTSTR lpszParameters,
   BOOL& bHandled
);
```

A virtual method that is invoked after the client has sent a command to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see Server Commands.

lpszResource

A null-terminated string that specifies the resource that the client has requested. Depending on the command issued, it may be a document, a folder or an executable script.

lpszParameters

A null-terminated string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

bHandled

An integer value that indicates whether the handler has processed the event. If the handler sets this parameter to a non-zero value, it indicates that the application has processed the command itself. If this parameter has a value of zero, then the server will perform the default action for the command. By default, this parameter has a value of zero.

Return Value

None.

Remarks

The **OnCommand** event handler is invoked after the client has sent a command to the server, but before the command has been processed. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This event handler is invoked for all commands issued by the client, including invalid or disabled commands. If the ***bHandled*** parameter is FALSE, the server will perform the default processing for the command. If the ***bHandled*** parameter is set to TRUE, the server will take no action. Note that if the event handler processes the command, it must call the **SendResponse** method to send a success or error response back to the client. If this is not done, the server will consider the command to be invalid and will send a 501 "not implemented" error by default.

It is not necessary to use this event handler to disable a command. The **EnableCommand** method can be used to enable or disable specific commands, and the **IsCommandEnabled** method can be

used to determine if a command is enabled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisconnectClient, EnableCommand, IsCommandEnabled, OnDisconnect, SendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnConnect Method

```
virtual void OnConnect(
   UINT nClientId,
   LPCTSTR lpszAddress
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A null-terminated string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and has accepted the connection. If the server was started with security enabled, the TLS handshake has been performed. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will terminate the client session prior to this event handler being invoked.

If this event handler is not implemented, the server will perform the default action of accepting the connection and waiting for the client to send a request for a document. To reject a connection, call the **DisconnectClient** method to terminate the client session.

Your server application should never make the assumption that for each **OnConnect** event there will be one command issued by the client. A client may issue multiple commands per session, and in some cases a client may send no commands to the server. If the client is using HTTP 1.0, connections will not be persistent. This means that the client will connect to the server, issue one command and then disconnect. For clients that use HTTP 1.1, multiple commands may be issued using a single connection. There is no guarantee that either the client or server will maintain a persistent connection, and either may request that the connection be closed after a command has completed.

If the client is a web browser, it is not unusual to see multiple, simultaneous connections being established with your server for a single request. Do not be concerned if you see multiple connections without any commands being issued. Browsers will do this to in anticipation of downloading additional assets (stylesheets, images, etc.) to improve performance and will typically close any unused connections after a few seconds. You may also see multiple requests for the file `/favicon.ico`. The browser will request this icon to display in the browser address bar and bookmarks. If the file does not exist, the browser ignores the error response from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisconnectClient, OnCommand, OnDisconnect,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnDisconnect Method

```
virtual void OnDisconnect(
   UINT nClientId
);
```

A virtual method that is invoked immediately before the client is disconnected from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked immediately before the client is disconnected from the server. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This is an advisory event and it is not required for the application to explicitly disconnect the client or perform any action. The event handler cannot prevent the client from disconnecting. Applications that implement a handler for this event should only use it to update any private data that was associated with the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

OnConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnDownload Method

```
virtual void OnDownload(
   UINT nClientId,
   LPCTSTR lpszFileName,
   DWORD dwTimeElapsed,
   ULARGE_INTEGER uiBytesCopied
);
```

A virtual method that is invoked after the client has successfully downloaded a file from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the local path name of the file that was downloaded from the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElased

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was downloaded by the client.

Return Value

None.

Remarks

The **OnDownload** event handler is invoked after the client has successfully downloaded a file from the server using the GET command. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OnCommand, OnUpload

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnError Method

```
virtual void OnConnect(
   UINT nClientId,
   UINT nEventId,
   DWORD dwError
);
```

A virtual method that is invoked when the event handler encountered an error when processing a client event.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see Server Event Constants.

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit TLS but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- A network error occurs when attempting to process a command issued by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetLastError, OnConnect, OnCommand

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnExecute Method

```
virtual void OnExecute(
   UINT nClientId,
   LPCTSTR lpszCommand,
   LPCTSTR lpszResource,
   LPCTSTR lpszParameters,
   DWORD dwExitCode
);
```

A virtual method that is invoked after the client has successfully executed a CGI program or script on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see Server Commands.

lpszResource

A null-terminated string that specifies the resource that the client has requested.

lpszParameters

A null-terminated string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

dwExitCode

An unsigned integer that specifies the exit code that was returned by the program.

Return Value

None.

Remarks

The **OnExecute** event handler is invoked after the client has successfully executed an external CGI program or script. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

External programs must be registered by the server application using the **RegisterProgram** method. To enable the use of scripts, the **RegisterHandler** method can be used to associate an executable program with a specific file extension.

The **GetProgramOutput** method can be used to obtain the unfiltered output from the external command, while the **GetProgramText** method will return filtered output from the program that contains only printable text characters.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProgramOutput, GetProgramText, RegisterHandler, RegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnResult Method

```
virtual void OnResult(
   UINT nClientId,
   LPCTSTR lpszCommand,
   LPCTSTR lpszResource,
   LPCTSTR lpszParameters,
   UINT nResultCode
);
```

A virtual method that is invoked after the server has processed a command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see Server Commands.

lpszResource

A null-terminated string that specifies the resource that the client has requested. Depending on the command issued, it may be a document, a folder or an executable script.

lpszParameters

A null-terminated string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

nResultCode

An integer value that specifies the result code that was sent to the client.

Return Value

None.

Remarks

The **OnResult** event handler is invoked after the server has processed a command issued by the client. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandResult, OnCommand

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnTimeout Method

```
virtual void OnTimeout(
   UINT nClientId,
   UINT nIdleTime,
   UINT nElapsed
);
```

A virtual method that is invoked after the client has exceeded the maximum allowed idle time.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nIdleTime

An unsigned integer value that specifies the current idle timeout period for the client in seconds.

nElapsed

An unsigned integer value that specifies the number of seconds that the client has been idle.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked after the client has has exceeded the maximum allowed idle time. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This event handler will be invoked prior to the client being disconnected from the server. This event will never occur while the server is returning data to the client. The **SetClientIdleTime** method can be used to change or refresh the idle timeout period for the session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

GetClientIdleTime, OnCommand, OnResult, SetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer::OnUpload Method

```
virtual void OnUpload(
   UINT nClientId,
   LPCTSTR lpszFileName,
   DWORD dwTimeElapsed,
   ULARGE_INTEGER uiBytesCopied
);
```

A virtual method that is invoked after the client has successfully uploaded a file to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the local path name of the file that was uploaded to the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElased

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was uploaded by the client.

Return Value

None.

Remarks

The **OnUpload** event handler is invoked after the client has successfully uploaded a file to the server using the PUT command. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OnCommand, OnDownload

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CHttpServer Data Structures

- HTTPCLIENTCREDENTIALS
- HTTPREQUEST
- HTTPRESPONSE
- HTTPSERVERTRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPCLIENTCREDENTIALS

The **HTTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _HTTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    UINT  nAuthType;
    UINT  nHostPort;
    TCHAR szHostName[HTTP_MAXHOSTNAME];
    TCHAR szUserName[HTTP_MAXUSERNAME];
    TCHAR szPassword[HTTP_MAXPASSWORD];
} HTTPCLIENTCREDENTIALS, *LPHTTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value is reserved for future use. This value will always be zero.

nAuthType

An unsigned integer value that identifies the authentication method.

nHostPort

The server port number.

szHostName

A null-terminated string that specifies the server host name.

szUserName

A null-terminated string that specifies the user name.

szPassword

A null-terminated string that specifies the user password.

Remarks

When an instance of this structure is passed to the **GetClientCredentials** method, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPREQUEST

The **HTTPREQUEST** structure provides information about the request made by the client.

```
typedef struct _HTTPREQUEST
{
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwVersion;
    DWORD  dwAccess;
    DWORD  dwLength;
    DWORD  dwReserved;
    UINT   nHostId;
    UINT   nHostPort;
    LPCSTR lpszCommand;
    LPCSTR lpszHostName;
    LPCSTR lpszUserName;
    LPCSTR lpszPassword;
    LPCSTR lpszResource;
    LPCSTR lpszParameters;
    LPCSTR lpszDirectory;
    LPCSTR lpszPathInfo;
    LPCSTR lpszProgram;
    LPCSTR lpszScriptFile;
    LPCSTR lpszLocalFile;
    LPCSTR lpszMediaType;
} HTTPREQUEST, *LPHTTPREQUEST;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that provides additional information about the request. It may be one or more of the following values:

Value	Description
HTTP_REQUEST_FLAG_PROTECTED	The resource that the client has requested is protected. The client should only be permitted to access the resource if the client session has been authenticated.
HTTP_REQUEST_FLAG_PROGRAM	The resource that the client has requested is a CGI program or an executable script. The output from the program or script should be returned to the client in the response. If this flag is specified, the ***lpszProgram*** member of the structure specifies the name of the program that should be executed.
HTTP_REQUEST_FLAG_SCRIPT	The resource that the client has requested is an executable script. This flag is set when the request URL is mapped to a registered script handler. The ***lpszProgram*** member specifies the name of the program that is responsible for executing the script, and the ***lpszScriptFile*** parameter specifies the full

	path to the script itself.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Value	Description
HTTP_VERSION_09	The client issued a GET command without specifying a protocol version.
HTTP_VERISON_10	The client issued a command that specified version 1.0 of the protocol.
HTTP_VERSION_11	The client issued a command that specified version 1.1 of the protocol.

dwAccess

An unsigned integer value that specifies the access permissions that were assigned to the resource. For a list of file access permissions, see User and File Access Constants.

dwLength

An unsigned integer value that will specify the number of bytes of request data provided by the client. If the client did not submit any data with the request, this member will have a value of zero.

dwReserved

An unsigned integer value that is reserved for future use.

nHostId

An integer value that identifies the virtual host that was specified by the client. This is based on the value of the Host request header included in the request. This value will be zero if a host name is not specified by the client, or does not match one of the virtual hosts that have been created.

nHostPort

An integer value that specifies the port number that the client used to establish the connection. This will be the same port number that was used when starting the server.

lpszCommand

A null-terminated string that specifies the command that was issued by the client. The command will always be in upper case. Refer to Hypertext Transfer Protocol Commands for a list of standard commands.

lpszHostName

A null-terminated string that identifies the hostname or IP address that the client used to establish the connection. This will typically correspond to the hostname assigned to the server, or to one of the virtual hosts that have been created.

lpszUserName

A null-terminated string that specifies the username provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszPassword

A null-terminated string that specifies the password provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszResource

A null-terminated string that specifies the URL path provided by the client.

lpszParameters

A null-terminated string that specifies any query parameters that were included in the request made by the client. If the URL did not include any query parameters, this member will be NULL.

lpszDirectory

A null-terminated string the specifies the full path to the root directory for the virtual host.

lpszPathInfo

A null-terminated string that specifies additional path information provided by the client when referencing an executable CGI program or script in the URL. For example, if a program is mapped to the virtual path "/orders/invoice" and the client requests "/orders/invoice/pdf/10001" this member will be the string "/pdf/10001". If there is no path information associated with the request, this member will be NULL.

lpszProgram

A null-terminated string that specifies the local path to the CGI program that should be executed to process the request made by the client. If this member is NULL, it indicates that the client has provided a URL that maps to a local file or directory.

lpszScriptFile

A null-terminated string that specifies the local path to the script file that will be executed by the handler. If this member is NULL, it indicates that the request URL was not mapped to registered script handler.

lpszLocalFile

A null-terminated string that specifies the local path to the directory or file name referenced by the URL. If the ***lpszPathInfo*** member is not NULL, then this member will point to a local file name based the path information appended to the root directory for the virtual host.

lpszMediaType

A null-terminated string that identifies the request data using the standard Internet media types defined in RFC 2046. It is used to designate the format for various types of content and has two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If the client did not submit any data with the request, this member will be NULL.

Remarks

This structure is used with the **ReceiveRequest** method and all members should be initialized to a value of zero, except for the ***dwSize*** member which should be initialized to size of the structure. Failure to properly initialize the structure will cause the method call to fail.

The value of the ***lpszLocalFile*** member depends on whether the client has requested a static document, or if the URL is mapped to a registered program or script handler. If the request is for a static document, then the ***dwFlags*** member will not have the HTTP_REQUEST_FLAG_PROGRAM bit flag set and the ***lpszLocalFile*** member will be the full path to the document. If the URL is mapped to a registered program or script, then the ***lpszLocalFile*** member will be the full path to the server root directory, If the request URL included additional path information, that will be appended to the root directory.

The value of the ***lpszScriptFile*** member is only defined if the ***dwFlags*** member has the HTTP_REQUEST_FLAG_SCRIPT bit flag set. This indicates that the request URL references a file that is an executable script with a handler that was registered using the **RegisterHandler** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

ReceiveRequest, SendResponse, HTTPRESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPRESPONSE

The **HTTPRESPONSE** structure provides additional information about the response to the client.

```
typedef struct _HTTPRESPONSE
{
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwVersion;
    DWORD   dwReserved;
    UINT    nExpires;
    UINT    nResultCode;
    LPCTSTR lpszReason;
    LPCTSTR lpszMediaType;
} HTTPRESPONSE, *LPHTTPRESPONSE;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that specifies one or more response flags. This member is reserved for future use and should always have a value of zero.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Value	Description
HTTP_VERSION_DEFAULT	The server should respond to the client using the same version specified in the request.
HTTP_VERISON_10	The server should respond to the client using version 1.0 of the protocol.
HTTP_VERSION_11	The server should respond to the client using version 1.1 of the protocol.

dwReserved

An unsigned integer value that is reserved for future use.

nExpires

An integer value that specifies the number of seconds until the client should consider any cached response data to be stale. If this value is zero, the default cache expiration time for the server will be used. The default cache expiration time is 7200 seconds (2 hours). The value of this structure member is ignored if the HTTP_RESPONSE_NOCACHE option is specified when the response is sent to the client.

nResultCode

An integer value that specifies the result code that should be sent in response to the client request. The result code is a three-digit number that indicates success or failure. For more information, refer to the **GetCommandResult** method.

lpszReason

A string that that describes the result code sent to the client. The description should be brief and

should not contain any formatting characters or HTML markup. This parameter may be NULL, in which case a default description of the result code will be used.

lpszMediaType

A string that specifies the Internet media type for the data that is being sent to the client as defined in RFC 2046. The format for the content type string consists of two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If this member is NULL, the library will attempt to automatically determine the appropriate media type.

Remarks

This structure is used with the **SendResponse** method and the ***dwSize*** member must be initialized to size of the structure. Failure to properly initialize the structure will cause the method call to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCommandResult, ReceiveRequest, SendResponse, HTTPREQUEST

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPSERVERTRANSFER

The **HTTPSERVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _HTTPSERVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} HTTPSERVERTRANSFER, *LPHTTPSERVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Value	Description
HTTP_FILE_READ	The file was opened for reading. This mode indicates that the client issued the GET command to download the contents of a file from the server to the client system. The ***szFileName*** member specifies the name of the local file on the server that was downloaded by the client.
HTTP_FILE_WRITE	The file was opened for writing. This mode indicates that the client issued the PUT command to upload the contents of a file from the client system to server. The ***szFileName*** member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a ULARGE_INTEGER union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file.

szFileName

A null-terminated string value that will contain the full path to the local file that was transferred. The ***dwFileAccess*** member determines whether the file name represents a file that was downloaded by

the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **GetTransferInfo** method, the ***dwSize*** member must be initialized to the size of the structure, otherwise the method will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

GetTransferInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify the security protocol and server certificate when accepting a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol Class Library

Determine if a remote host is reachable and how packets of data are routed to that system.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CIcmpClient
File Name	CSICMV12.DLL
Version	12.0.3024.1286
LibID	66D5E749-831F-40CF-9B36-77A849C6C0E5
Import Library	CSICMV12.LIB
Dependencies	None
Standards	RFC 792

Overview

The Internet Control Message Protocol (ICMP) is commonly used to determine if a remote host is reachable and how packets of data are routed to that system. Users are most familiar with this protocol as it is implemented in the ping and traceroute command line utilities. The ping command is used to check if a system is reachable and the amount of time that it takes for a packet of data to make a round trip from the local system, to the remote host and then back again. The traceroute command is used to trace the route that a packet of data takes from the local system to the remote host, and can be used to identify potential problems with overall throughput and latency. The library can be used to build in this type of functionality in your own applications, giving you the ability to send and receive ICMP echo datagrams in order to perform your own analysis.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on

ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol Class Methods

Class	Description
CIcmpClient	Constructor which initializes the current instance of the class
~CIcmpClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of network-level calls to the trace log
Echo	Send one or more echo datagrams to the specified host
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of network function calls to a file
FormatAddress	Convert a numeric IP address to a string
FreezeEvents	Suspend or resume event handling by the calling process
GetErrorString	Return a description for the specified error code
GetFirstHost	Return the first host from the traceroute
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the current host IP address
GetHostName	Return the current host name
GetLastError	Return the last error code
GetNextHost	Return the next host from the traceroute
GetPacketSize	Return the ICMP datagram packet size
GetRecvCount	Return the number of packets received
GetSendCount	Return the number of packets sent
GetSequenceId	Return the current sequence identifier
GetTimeout	Return the number of milliseconds until an operation times out
GetTimeToLive	Return the current time-to-live for the ICMP datagram
GetTripTime	Return the current trip time statistics
IcmpEventProc	Callback method that processes events generated by the client
IsBlocking	Determine if the client is blocked, waiting for information
IsInitialized	Determine if the class has been successfully initialized

RecvEcho	Read an ICMP datagram returned by the remote host
RegisterEvent	Register an event callback function
Reset	Reset the current client state
ResolveAddress	Resolve an IP address into a fully qualified host name
SendEcho	Send an ICMP datagram to the specified host
SetHostAddress	Set the IP address of the host to receive the next datagram
SetHostName	Set the name of the host to receive the next datagram
SetLastError	Set the last error code
SetPacketSize	Set the ICMP datagram packet size
SetSequenceId	Set the sequence identifier for the next datagram
SetTimeout	Set the number of milliseconds until an operation times out
SetTimeToLive	Set the time-to-live for the next datagram
ShowError	Display a message box with a description of the specified error
TraceRoute	Trace the route from the local system to a remote host

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::CIcmpClient Method

```
CIcmpClient();
```

The **CIcmpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **IcmpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CIcmpClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::~CIcmpClient

```
~CIcmpClient();
```

The **CIcmpClient** destructor releases resources allocated by the current instance of the **CIcmpClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CIcmpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CIcmpClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

AttachHandle, Cancel, CloseHandle, CreateHandle, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::CloseHandle Method

```
INT CloseHandle(
   HCLIENT hClient
);
```

The **CloseHandle** method closes the socket and releases the memory allocated for the client session.

Parameters

hClient

A handle to the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CIcmpClient, CreateHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::CreateHandle Method

```
HCLIENT CreateHandle(
   UINT nPacketSize,
   UINT nTimeToLive,
   DWORD dwTimeout,
   DWORD dwReserved,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **CreateHandle** method creates a client handle for sending and receiving ICMP echo datagrams. If an event notification window is specified, the client will be notified when a network event occurs.

Parameters

nPacketSize

An unsigned integer which specifies the default packet size used when generating ICMP echo datagrams. The minimum packet size is 32 bytes and the maximum size is 65,535 bytes.

nTimeToLive

An unsigned integer which specifies the default time-to-live for ICMP echo datagrams. This determines the maximum number of times that a packet will be routed from one system to another while en route to its destination. The minimum time-to-live value is 1, the maximum is 255. The recommend value for this parameter is 255, and typical applications should use a time-to-live value of at least 30.

dwTimeout

An unsigned integer which specifies the maximum number of milliseconds to wait before the current operation times out.

dwReserved

A reserved parameter. This value should always be zero.

hEventWnd

The handle to an asynchronous notification window. This window receives messages which notify the client when asynchronous network events occur. If asynchronous event notification is not required, this parameter may be NULL.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a handle to the client session. If the method fails, the return value is INVALID_CLIENT. To get extended error information, call **GetLastError**.

Remarks

The **CreateHandle** method creates a client handle that is used with subsequent calls to the library. This library uses a special type of socket called a raw socket, which is created to send and receive ICMP echo datagrams. Raw socket support is optional under the Windows Sockets specification, and may not be available if a non-standard networking libraries are used or may only be available to privileged accounts.

If the *hEventWnd* parameter is not NULL, the client operates in asynchronous mode and messages will be posted to the notification window when a network event occurs. When a message is posted to the window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. TThe client application may attempt to retry the operation or close the handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

The ability to create and send ICMP echo datagrams is limited to privileged users. Non-administrator users will receive an error if they attempt to create a client handle. On Windows NT it is possible to disable this security check by creating or modifying the system registry. Microsoft Knowledge Base article 195445 has additional information and instructions for making this change.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Sockets 2.0 update or later.
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

CIcmpClient, CloseHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::Echo Method

```
INT Echo(
   LPCTSTR lpszHostName,
   UINT nIterations,
   UINT nPacketSize,
   DWORD dwTimeout,
   LPICMPTIME lpTime
);

INT Echo(
   LPCTSTR lpszHostName,
   LPICMPTIME lpTime
);
```

The **Echo** method sends one or more ICMP echo datagrams, collecting information about the reliability and latency of a connection between the local and remote host.

Parameters

lpszHostName

A null-terminated string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nIterations

An unsigned integer value which specifies the number of echo datagrams that will be sent to the remote host. The minimum value for this parameter is 1 and the maximum value is 512.

nPacketSize

An unsigned integer value which specifies the size of the echo datagram in bytes. The minimum size is 1 byte and the maximum size is 65,535 bytes. It is recommended that most applications use the minimum size of 32 bytes for this parameter.

dwTimeout

An unsigned integer which specifies the number of milliseconds the method will wait for a response to an echo datagram.

lpTime

A pointer to an ICMPTIME structure which will contain the minimum, maximum and average round trip times for the echo datagrams sent and received.

Return Value

If the method succeeds, the return value is the number of replies received from the remote host. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Echo** method sends a series of ICMP echo datagrams to the specified host, providing a simplified interface for pinging a remote system. If the method returns the same value as the number of iterations, then replies were received for all of the echo datagrams that were sent. This would typically indicate that the client can establish a reliable connection to the server; the values returned in the ICMPTIME structure provide information on the latency between the two hosts. Higher average time values would indicate greater latency and reduced throughput between the systems. If the method returns a value less than the specified number of iterations, this indicates that replies were not received for one or more of the echo datagrams. This may indicate that there are problems routing

data between the local and remote host. A return value of zero indicates that there was no response to the echo datagrams. The remote host may not exist or may not be available.

The failure for a host to respond to an ICMP echo datagram may not indicate a problem with the remote system. In some cases, a router between the local and remote host may be malfunctioning or discarding the datagrams. Systems can also be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

CreateHandle, GetTripTime, RecvEcho, SendEcho, TraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
ICMP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::FormatAddress Method

```
INT FormatAddress(
   DWORD dwAddress,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT FormatAddress(
   DWORD dwAddress,
   CString& strAddress
);
```

The **FormatAddress** method converts a numeric IP address to a null-terminated string.

Parameters

dwAddress

A unsigned 32-bit integer which specifies the IP address to be converted into a string.

lpszAddress

A null-terminated string which will contain the converted IP address in dot-notation. This string should be at least 16 characters in length. An alternate form of this method accepts a **CString** object which will contain the IP address in dotted notation when the method returns.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the method succeeds, the return value is the length of the string buffer. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, ResolveAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetFirstHost Method

```
BOOL GetFirstHost(
   ICMPTRACE& icmpTrace
);
```

The **GetFirstHost** method returns information about the first host between the local system and the remote host specified in the call to the **TraceRoute** method.

Parameters

icmpTrace

An ICMPTRACE structure which contains information about the intermediate host.

Return Value

If the method succeeds, the return value is non-zero. If the return value is zero, this indicates that no host information is available. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstHost** method is used in conjunction with the **GetNextHost** method to enumerate all of the intermediate hosts between the local system and the remote host specified when the **TraceRoute** method was called.

Example

```
BOOL bResult = pClient->TraceRoute(strAddress);
if (bResult)
{
    ICMPTRACE icmpTrace;

    bResult = pClient->GetFirstHost(icmpTrace);
    while (bResult)
    {
        // The icmpTrace structure contains information about the
        // intermediate host in the traceroute
        CString strHostName;
        pClient->ResolveAddress(icmpTrace.dwHostAddress, strHostName);

        bResult = pClient->GetNextHost(icmpTrace);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetNextHost, ResolveAddress, TraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetHostAddress Method

```
INT GetHostAddress(
   LPTSTR lpszHostAddress,
   INT cbHostAddress
);

INT GetHostAddress(
   CString& strHostAddress
);
```

The **GetHostAddress** copies the IP address of the current host into the specified buffer as a string using dot-notation.

Parameters

lpszHostAddress

The buffer that will contain the IP address of the current remote host in dot-notation. This buffer should be at least 16 characters in length.

cbHostAddress

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the method succeeds, the return value is the length of the address string. If the return value is zero, this indicates that no remote host has been specified. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, GetHostName, ResolveAddress, SetHostAddress, SetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetHostName Method

```
INT GetHostName(
   LPTSTR lpszHostName,
   INT cbHostName
);

INT GetHostName(
   CString& strHostName
);
```

The **GetHostName** copies the name of the current host into the specified buffer.

Parameters

lpszHostName

The buffer that will contain the name of the current remote host. This buffer should be at least 64 characters in length.

cbHostName

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the method succeeds, the return value is the length of the host name string. If the return value is zero, this indicates that no remote host has been specified. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, GetHostAddress, ResolveAddress, SetHostAddress, SetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetNextHost Method

```
BOOL GetFirstHost(
   ICMPTRACE& icmpTrace
);
```

The **GetNextHost** method returns information about the next host between the local system and the remote host specified in the call to the **TraceRoute** method.

Parameters

icmpTrace

An ICMPTRACE structure which contains information about the intermediate host.

Return Value

If the method succeeds, the return value is non-zero. If the return value is zero, this indicates that no host information is available. To get extended error information, call **GetLastError**.

Remarks

The **GetNextHost** method is used in conjunction with the **GetFirstHost** method to enumerate all of the intermediate hosts between the local system and the remote host specified when the **TraceRoute** method was called.

Example

```
BOOL bResult = pClient->TraceRoute(strAddress);
if (bResult)
{
    ICMPTRACE icmpTrace;

    bResult = pClient->GetFirstHost(icmpTrace);
    while (bResult)
    {
        // The icmpTrace structure contains information about the
        // intermediate host in the traceroute
        CString strHostName;
        pClient->ResolveAddress(icmpTrace.dwHostAddress, strHostName);

        bResult = pClient->GetNextHost(icmpTrace);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstHost, ResolveAddress, TraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetPacketSize Method

```
UINT GetPacketSize();
```

The **GetPacketSize** method returns the size of the ICMP echo datagram that will be sent to the remote host. The minimum packet size is 1 byte, the maximum packet size is 65,535 bytes.

Parameters

None.

Return Value

If the method succeeds, the return value is the size of the datagram. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetRecvCount, GetSendCount, GetSequenceId, GetTimeToLive, GetTripTime, SetPacketSize, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetRecvCount Method

```
INT GetRecvCount();
```

The **GetRecvCount** method returns the number of replies sent to the client from the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of replies. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetPacketSize, GetSendCount, GetSequenceId, GetTimeToLive, GetTripTime, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetSendCount Method

```
INT GetSendCount();
```

The **GetSendCount** method returns the number of datagrams sent to the remote host by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of datagrams sent by the client. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetPacketSize, GetRecvCount, GetSequenceId, GetTimeToLive, GetTripTime, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetSequenceId Method

```
INT GetSequenceId();
```

The **GetSequenceId** method returns the sequence identifier for the last ICMP echo datagram received by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the sequence identifier. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetPacketSize, GetRecvCount, GetSendCount, GetTimeToLive, GetTripTime, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

CreateHandle, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetTimeToLive Method

```
INT GetTimeToLive();
```

The **GetTimeToLive** method returns the time-to-live for the last ICMP echo datagram received by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the time-to-live for the last datagram. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded. The default value is 255.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetPacketSize, GetRecvCount, GetSendCount, GetSequenceId, GetTripTime, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::GetTripTime Method

```
INT GetTripTime(
   LPICMPTIME lpTime
);
```

The **GetTripTime** returns round-trip statistics for the datagrams sent to the current remote host.

Parameters

lpTime

A pointer to an ICMPTIME data structure which will contain the round-trip statistics for the current remote host.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetPacketSize, GetRecvCount, GetSendCount, GetSequenceId, GetTimeToLive, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

CIcmpClient, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::RecvEcho Method

```
INT RecvEcho();
```

The **RecvEcho** method reads the reply to an ICMP echo datagram generated by the client. This method should only be called by asynchronous client sessions in response to an event notification that a datagram has been received.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetRecvCount, GetSequenceId, GetTimeToLive, SendEcho

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   ICMPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
ICMP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **IcmpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **IcmpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, IcmpEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client session, clearing the packet trip statistics, time-to-live, sequence identifier, send and receive counts.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

SetHostAddress, SetHostName, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::ResolveAddress Method

```
INT ResolveAddress(
   DWORD dwAddress,
   LPTSTR lpszHostName,
   INT nMaxLength
);

INT ResolveAddress(
   DWORD dwAddress,
   CString& strHostName
);
```

The **ResolveAddress** method resolves a numeric IP address into a fully qualified domain name.

Parameters

dwAddress

The IP address to be resolved, specified as an unsigned 32-bit integer in network byte order.

lpszHostName

A buffer that will contain a null-terminated string that specifies the fully qualified domain name for the host. It is recommended that this buffer be at least 64 characters in length.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the host name. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ResolveAddress** method is used to perform a reverse DNS lookup, converting a numeric IP address into a fully qualified domain name. This is useful for methods like **TraceRoute**, which return host addresses as numeric values in network byte order. If the reverse DNS lookup fails because there is no PTR record for the given IP address, a printable form of the address in dotted notation will be returned in the string buffer.

Calling this method will cause the thread to block until the IP address is resolved, or until the DNS query times out because there is no reverse record for the address. In some cases, a reverse lookup can take several seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetHostAddress, GetHostName, SetHostAddress, SetHostName, TraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SendEcho Method

```
INT SendEcho();
```

The **SendEcho** method sends an ICMP echo datagram to the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

For asynchronous client sessions, this method returns immediately. Otherwise, the client enters a blocking state and waits for a reply from the remote host. The method returns when a reply has been received, or the operation times-out.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Echo, GetSendCount, RecvEcho, SetSequenceId, SetTimeToLive, TraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetHostAddress Method

```
INT SetHostAddress(
   DWORD dwAddress
);

INT SetHostAddress(
   LPCTSTR lpszAddress
);
```

The **SetHostAddress** method specifies the IP address of the host to receive an ICMP echo datagram. If the address specifies a new host, the current client statistics are reset.

Parameters

dwAddress

The IP address of the remote host as a 32-bit integer value, specified in network byte order. In an alternate form of this method, a pointer to a string which specifies the IP address in dotted notation may also be used.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

To specify a remote host name, use the **SetHostName** method instead.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetHostAddress, GetHostName, SetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetHostName Method

```
INT SetHostName(
   HCLIENT hClient,
   LPCTSTR lpszHostName
);
```

The **SetHostName** method specifies the name of the host to receive an ICMP echo datagram. If the name specifies a new host, the current client statistics are reset.

Parameters

hClient

A handle to the client session.

lpszHostName

A null-terminated string which specifies the name, or IP address in dot-notation, of the remote host.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, SetHostAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetPacketSize Method

```
UINT SetPacketSize(
   UINT nPacketSize
);
```

The **SetPacketSize** method sets the size of the ICMP datagram packet that is sent to the remote host.

Parameters

nPacketSize

Size of the ICMP datagram packet in bytes. The minimum packet size is 1 byte and the maximum packet size is 65,535 bytes.

Return Value

If the method succeeds, the return value is the previous ICMP datagram packet size. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Note that packet sizes over 512 bytes may not be supported by your networking software or configuration. It is recommended that most applications use the minimum packet size of 32 bytes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetPacketSize, SetSequenceId, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetSequenceId Method

```
INT SetSequenceId(
   INT nSequenceId
);
```

The **SetSequenceId** method sets the sequence identifier for the next ICMP echo datagram sent by the client. The default sequence identifier for the first datagram is one.

Parameters

nSequenceId

The sequence identifier for the next datagram sent by the client.

Return Value

If the method succeeds, the return value is the previous sequence identifier. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetPacketSize, GetRecvCount, GetSendCount, GetSequenceId, GetTimeToLive, GetTripTime, SetPacketSize, SetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::SetTimeToLive Method

```
INT SetTimeToLive(
   INT nTimeToLive
);
```

The **SetTimeToLive** method sets the maximum time-to-live for the next ICMP datagram sent by the client.

Parameters

nTimeToLive

The time-to-live value for the next ICMP echo datagram.

Return Value

If the method succeeds, the return value is the previous time-to-live value. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded.

Calling this method changes the default TTL value for all subsequent ICMP datagrams sent by the library, with the default value being 255. Note that not all Windows Sockets implementations support setting the time-to-live value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

GetPacketSize, GetRecvCount, GetSendCount, GetSequenceId, GetTimeToLive, GetTripTime, SetPacketSize, SetSequenceId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CIcmpClient::TraceRoute Method

```
BOOL TraceRoute(
   LPCTSTR lpszHostName,
   UINT nMaxHops,
   DWORD dwTimeout
);
```

The **TraceRoute** method sends a series of ICMP echo datagrams to trace the route taken from the local system to the remote host.

Parameters

lpszHostName

A null-terminated string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nMaxHops

An unsigned integer which specifies the maximum number of routers the datagram will be forwarded through (the number of hops) to the remote host. The minimum value is 1 and the maximum value is 255. It is recommended that most applications specify a value of at least 30.

dwTimeout

An unsigned integer which specifies the number of milliseconds the method will wait for a response to an echo datagram.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **TraceRoute** method sends a series of ICMP echo datagrams to the specified host, adjusting the time-to-live value to determine the intermediate hosts that route the packet. Use the **GetFirstHost** and **GetNextHost** methods to enumerate the hosts returned by the method.

It is important to note that the failure of an intermediate host to respond to an ICMP echo datagram may not indicate a problem with the remote system. Systems can be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Example

```
BOOL bResult = pClient->TraceRoute(strAddress);
if (bResult)
{
    ICMPTRACE icmpTrace;

    bResult = pClient->GetFirstHost(icmpTrace);
    while (bResult)
    {
```

```
        // The icmpTrace structure contains information about the
        // intermediate host in the traceroute
        CString strHostName;
        pClient->ResolveAddress(icmpTrace.dwHostAddress, strHostName);

        bResult = pClient->GetNextHost(icmpTrace);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

Echo, GetFirstHost, GetNextHost, ResolveAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol Data Structures

- ICMPTIME
- ICMPTRACE
- INTERNET_ADDRESS
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ICMPTIME

This structure is used by the GetTripTime method to return the round-trip times of an ICMP datagram.

```
typedef struct _ICMPTIME {
    DWORD dwTripAverage;
    DWORD dwTripMaximum;
    DWORD dwTripMinimum;
    DWORD dwTripTime;
} ICMPTIME, *LPICMPTIME;
```

Members

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the current host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to current host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the current host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the current host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ICMPTRACE

This structure is used by the GetFirstHost and GetNextHost methods to return the route of an ICMP datagram.

```
typedef struct _ICMPTRACE
{
    UINT        nDistance;
    DWORD       dwHostAddress;
    DWORD       dwTripAverage;
    DWORD       dwTripMaximum;
    DWORD       dwTripMinimum;
} ICMPTRACE, *LPICMPTRACE;
```

Members

nDistance

The distance from the local host to the remote host for this route.

dwHostAddress

An unsigned integer which specifies the IP address of the remote host in network byte order.

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to specified host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the specified host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
     INT    ipFamily;
     BYTE   ipNumber[16];
}  INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol Class Library

Manage email messages and mailboxes on a mail server.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CImapClient
File Name	CSMAPV12.DLL
Version	12.0.3024.1286
LibID	8A55FC0D-5E12-4932-9E86-1F88C1DF5024
Import Library	CSMAPV12.LIB
Dependencies	None
Standards	RFC 3501

Overview

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP.

The SocketTools IMAP library implements the current standard for this protocol, and provides methods to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The API is designed as a superset of the Post Office Protocol API, so developers who are used to working with the POP3 library will find the IMAP library very easy to integrate into an existing application.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol Class Methods

Class	Description
CImapClient	Constructor which initializes the current instance of the class
~CImapClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
CheckMailbox	Check the current mailbox for new messages
CloseMessage	Close the current message
Command	Send a command to the server
Connect	Connect to the specified server
CopyMessage	Copy a message from the current mailbox to another mailbox
CreateMailbox	Create a new mailbox on the server
CreateMessage	Create a new message in the specified mailbox
CreateSecurityCredentials	Allocate a structure to establish client security credentials
DecodeMailboxName	Decode a UTF-7 encoded mailbox name using the specified code page
DeleteMailbox	Delete the specified mailbox from the server
DeleteMessage	Delete the specified message from the mailbox
DeleteSecurityCredentials	Delete the specified client security credentials
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable the event notification mechanism
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable the client event notification mechanism
EnableTrace	Enable logging of network function calls to a file
EncodeMailboxName	Encode a mailbox name using modified UTF-7 encoding
EnumMessages	Enumerate the messages in the current mailbox
ExamineMailbox	Select the specified mailbox in read-only mode
ExpungeMailbox	Remove messages that have been marked for deletion from the current mailbox
FreezeEvents	Suspend and resume event handling by the client
GetCapability	Return a string which identifies the capabilities of the server
GetCurrentMailbox	Return the name of the currently selected mailbox

GetDeletedMessages	Return the messages that have been marked for deletion
GetErrorString	Return a description for the specified error code
GetFirstMailbox	Return the first mailbox according to specified criteria
GetHandle	Return the client handle used by this instance of the class
GetHeaderValue	Return the value of the specified header field
GetIdleThreadId	Return the ID of the thread created to monitor the client session
GetLastError	Return the last error code
GetMailboxDelimiter	Get the path delimiter for the specified mailbox hierarchy
GetMailboxSize	Return the size of the specified mailbox
GetMailboxStatus	Return the status of the specified mailbox
GetMailboxUID	Return the unique identifier for the specified mailbox
GetMessage	Retrieve the specified message from the server
GetMessageCount	Return the number of messages available in the mailbox
GetMessageFlags	Return the status flags for the specified message
GetMessageHeaders	Retrieve the specified message header from the server
GetMessageId	Return the message ID string for the specified message
GetMessageParts	Return the number of MIME message parts in the specified message
GetMessageSender	Return the address of the message sender
GetMessageSize	Return the size of the specified message
GetMessageUID	Return the unique identifier for the specified message
GetNewMessages	Return a list of the new messages in the current mailbox
GetNextMailbox	Return the next mailbox name on the server
GetResultCode	Return the result code from the previous command
GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current status of the client
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
GetUnseenMessages	Return a list of messages from the current mailbox that have not been read
Idle	Enables mailbox status monitoring for the client session
ImapEventProc	Callback method that processes events generated by the client
ImapIdleProc	Callback function that receives update notifications from the server
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized

IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
Login	Login to the server
OpenMessage	Open the specified message for reading on the server
Read	Read data returned by the server
RegisterEvent	Register an event handler for the specified event
RenameMailbox	Rename the specified mailbox
ReselectMailbox	Reselect the current mailbox and return updated status information
SearchMailbox	Search the mailbox according to specified criteria
SelectMailbox	Select the specified mailbox in read-write mode
SetLastError	Set the last error code for the current thread
SetMessageFlags	Set one or more status flags for the specified message
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
StoreMessage	Store the contents of a message to the specified file
SubscribeMailbox	Subscribe to the specified mailbox
UndeleteMessage	Undelete the specified message from the current mailbox
UnselectMailbox	Unselect the current mailbox and expunge any deleted messages
UnsubscribeMailbox	Unsubscribe from the specified mailbox
Write	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::~CImapClient

```
~CImapClient();
```

The **CImapClient** destructor releases resources allocated by the current instance of the **CImapClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CImapClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CImapClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CheckMailbox Method

```
INT CheckMailbox(
   UINT *lpnMessages,
   UINT *lpnUnseen
);
```

The **CheckMailbox** method requests that the server create a checkpoint of the currently selected mailbox, and returns the current number of messages and unseen messages.

Parameters

lpnMessages

A pointer to an unsigned integer value which will contain the number of messages in the currently selected mailbox when the method returns.

lpnUnseen

A pointer to an unsigned integer value which will contain the number of unseen messages in the currently selected mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the client requests a checkpoint, the server may perform implementation-dependent housekeeping for that mailbox, such updating the mailbox on disk with the current state of the mailbox in memory. On some systems this command has no effect other than to update the client with the current number of messages in the mailbox.

This method actually sends two IMAP commands. The first is the CHECK command, followed by the NOOP command to poll for any new messages that have arrived. In addition to polling the server for new messages, this command can also be used to ensure the idle timer on the server does not expire and force a disconnect from the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

CreateMailbox, GetFirstMailbox, GetMailboxStatus, GetNextMailbox, RenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CImapClient Method

```
CImapClient();
```

The **CImapClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **ImapInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CImapClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CloseMessage Method

```
INT CloseMessage();
```

The **CloseMessage** method closes the current message.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CloseMessage** method closes the current message. If there is any remaining data left to be read from the message, it will be read and discarded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

CreateMessage, OpenMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Command Method

```
INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **Command** method sends a command to the server. This method is typically used for site-specific commands not directly supported by the API.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the method succeeds, it returns an IMAP result code. If the command was successful, it returns IMAP_RESULT_OK. A return value of IMAP_RESULT_CONTINUE indicates the command was accepted and the caller should proceed with the next command. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. The current version of the protocol which is supported by this library is version 4rev1 as defined in RFC 3501.

Use the **GetResultCode** method to determine the result of the command that was sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszHostName,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the IMAP server.

nRemotePort

The port number the client should use to establish the connection. A value of zero specifies that default port 143 should be used, which is the standard port number assigned to the IMAP service. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
IMAP_OPTION_IDENTIFY	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
IMAP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
IMAP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
IMAP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the

	server support the TLS protocol. This option is the same as specifying IMAP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
IMAP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the STARTTLS command.
IMAP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
IMAP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
IMAP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
IMAP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the asynchronous notification window. This window receives messages which notify the client of various asynchronous client events that occur. Specifying this parameter and a message identifier causes the connection to be non-blocking. If this parameter is NULL, then a blocking connection is established.

uEventMsg

The message identifier that is used when an asynchronous client event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the IMAP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, Login, RegisterEvent, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CopyMessage Method

```
INT CopyMessage(
   UINT nMessageId,
   LPCTSTR lpszMailbox
);
```

The **CopyMessage** method copies a message from the current mailbox to the specified mailbox. The message is appended to the mailbox, and the message flags and internal date are preserved.

Parameters

nMessageId

The message identifier which specifies which message is to be copied to the mailbox. This value must be greater than zero and specify a valid message number.

lpszMailbox

A null-terminated string which specifies the name of the mailbox that the message will be copied to. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

Return Value

If the method succeeds, it returns a value of zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the mailbox does not exist, the method will fail. To create a new mailbox, use the **CreateMailbox** method. A message can be copied within the same mailbox, in which case the server may flag it as a new message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CreateMailbox Method

```
INT CreateMailbox(
   LPCTSTR lpszMailbox
);
```

The **CreateMailbox** method creates a new mailbox on the server.

Parameters

lpszMailbox

A null-terminated string which specifies the new mailbox to be created.

Return Value

If the method succeeds, it returns a value of zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the mailbox name is suffixed with the server's hierarchy delimiter, this indicates to the server that the client intends to create mailbox names under the specified name in the hierarchy. If superior hierarchical names are specified in the mailbox name, then the server may automatically create them as needed. For example, if the mailbox name "Mail/Office/Projects" is specified and "Mail/Office" does not exist, it may be automatically created by the server.

The special mailbox name INBOX is reserved, and cannot be created. It is recommended that mailbox names only consist of printable ASCII characters, and the special characters "*" and "%" should be avoided. If you need to create a mailbox which contains Unicode characters or symbols, your application should be compiled to use the Unicode character set to ensure that the mailbox name is encoded correctly. If you do not use Unicode, this method will only accept mailbox names using either modified UTF-7 encoding or UTF-8 encoding.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMailbox, GetFirstMailbox, GetNextMailbox, RenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CreateMessage Method

```
INT CreateMessage(
   LPCTSTR lpszMailbox,
   LPBYTE lpMessage,
   DWORD dwMessageSize,
   DWORD dwFlags
);

INT CreateMessage(
   LPCTSTR lpszMailbox,
   LPCTSTR lpszMessage,
   DWORD dwFlags
);
```

The **CreateMessage** method creates a message, appending it to the contents of the specified mailbox.

Parameters

lpszMailbox

A null-terminated string which specifies the name of the mailbox that the message will be created in. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

lpMessage

The buffer which contains the data for the message to be created. An alternate form of the method accepts a pointer to a string.

dwMessageSize

An unsigned integer value which specifies the size of the message data in bytes.

dwFlags

An unsigned integer that specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message is not completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message is flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the mailbox does not exist, the method will fail. To create a new mailbox, use the **CreateMailbox** method. If the message is created in the mailbox that is currently selected, the server may flag it as recent.

This method is typically used by applications to store messages which have already been sent to a user. After a message has been delivered using the SMTP protocol, that same message may be created in a mailbox on the IMAP server so that the user has access to those messages.

If the Unicode version of this method is called and the ***dwMessageSize*** parameter is zero, the ***lpMessage*** parameter must point to a wide string terminated with a null character. The string will be automatically converted to UTF-8 when the message is created. In all other cases, the ***lpMessage*** parameter is treated as a pointer to a byte array which contains the message contents and the ***dwMessageSize*** parameter specifies the number of bytes in the array.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CopyMessage, DeleteMessage, GetMessageFlags, SetMessageFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              IMAP_PORT_SECURE,
                              IMAP_TIMEOUT,
                              IMAP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DecodeMailboxName Method

```
INT DecodeMailboxName(
   LPCSTR lpszMailboxName,
   LPTSTR lpszDecodedName,
   INT nMaxLength,
   INT nCodePage
);

INT DecodeMailboxName(
   LPCSTR lpszMailboxName,
   CString& strDecodedName,
   INT nCodePage
);
```

The **DecodeMailboxName** method decodes a UTF-7 encoded mailbox name using the specified code page.

Parameters

lpszMailboxName

A null-terminated string which contains the UTF-7 encoded name of the mailbox. The encoded string must use the modified UTF-7 format defined in the IMAP4 protocol standard. This parameter cannot be a NULL pointer or an empty string.

lpszDecodedName

A null-terminated string buffer which will contain the decoded mailbox name. This parameter cannot be NULL and must be large enough to contain the entire name. An alternate form of the method accepts a **CString** object which will contain the decoded mailbox name when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the *lpszDecodedName* string buffer. This value must be greater than zero and the buffer must be large enough to store the entire name.

nCodePage

An optional integer value which specifies the code page used when decoding the mailbox name. If the application does not use Unicode, this parameter can be any valid code page. The default value is CP_ACP which specifies the current active code page for the locale. Applications which are built using Unicode should always use the default value.

Return Value

If the method succeeds, it returns the number of characters in the decoded mailbox name. If an error occurs, the method returns zero. To get extended error information, call **GetLastError**.

Remarks

Mailbox names which contain non-ASCII characters must be converted to a modified version of UTF-7 encoding as defined by the IMAP4 protocol. This encoding allows applications to reference mailbox names which contain Unicode characters and provides support for languages other than English. This method is used to convert a UTF-7 encoded mailbox name using the current locale, primarily for display purposes.

For example, if the *lpszMailboxName* parameter specifies the string **"&A5QDvwO6A7kDvAOu-"**, this

method will convert it to the Greek word **"Δοκιμή"** and return that value in the *lpszDecodedName* buffer. It is recommended you use the Unicode version of this method whenever possible to ensure the mailbox name is decoded correctly.

If the application is not built using Unicode, the decoded mailbox name will be converted using the code page specified by the *nCodePage* parameter. If the code page does not support the characters used in the mailbox name, this can result in an incorrect conversion.

This method should not be used to convert strings between Unicode and UTF-7. The encoding format is specifically intended for use with the IMAP4 protocol and should never be used for general purpose encoding or decoding. If you need to convert a UTF-7 encoded string to Unicode, use the Win32 API function **MultiByteToWideChar**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, EncodeMailboxName, GetFirstMailbox, GetNextMailbox, RenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DeleteMailbox Method

```
INT DeleteMailbox(
   LPCTSTR lpszMailbox
);
```

The **DeleteMailbox** method deletes a mailbox on the server.

Parameters

lpszMailbox

A null-terminated string which specifies the mailbox to be deleted.

Return Value

If the method succeeds, it returns value of zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A mailbox cannot be deleted if it contains inferior hierarchical names and has the IMAP_FLAG_NOSELECT attribute. On most systems this is the case when the mailbox name references a directory on the server, and that directory contains other subdirectories or mailboxes. To remove the mailbox, you must first delete any child mailboxes that exist.

If the mailbox that is deleted is the currently selected mailbox, it will be automatically unselected and any messages marked for deletion will be expunged before the mailbox is removed. If the delete operation fails, the client will remain in an unselected state until either the **ExamineMailbox** or **SelectMailbox** method is called.

The special mailbox name INBOX is reserved, and cannot be deleted.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, GetFirstMailbox, GetNextMailbox, RenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DeleteMessage Method

```
INT DeleteMessage(
   UINT nMessageId
);
```

The **DeleteMessage** method marks the specified message for deletion from the current mailbox.

Parameters

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method only marks the message for deletion. The message is not actually deleted until the mailbox is expunged or another mailbox is selected. This method will return an error if the current mailbox is in read-only mode, such as if it was selected using the **ExamineMailbox** method.

It is important to note that unlike the POP3 protocol, a message that is marked for deletion is still accessible on the IMAP server until the mailbox is expunged. This means, for example, that a deleted message can still be retrieved using the **GetMessage** method.

To determine if a message has been marked for deletion, use the **GetMessageFlags** method and check if the IMAP_FLAG_DELETED bit flag has been set. To list all of the deleted messages in the current mailbox, use the **GetDeletedMessages** method.

To remove the deletion flag from the message, use the **UndeleteMessage** method. To prevent all messages in the current mailbox from being expunged, use the **ReselectMailbox** method to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetDeletedMessages, GetMessage, GetMessageCount, GetMessageFlags, ReselectMailbox, UndeleteMessage, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This

	event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
IMAP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
IMAP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
IMAP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::EncodeMailboxName Method

```
INT EncodeMailboxName(
   LPCTSTR lpszMailboxName,
   LPTSTR lpszEncodedName,
   INT nMaxLength,
   INT nCodePage
);

INT EncodeMailboxName(
   LPCTSTR lpszMailboxName,
   CString& strEncodedName,
   INT nCodePage
);
```

The **EncodeMailboxName** method encodes a mailbox name using modified UTF-7 encoding.

Parameters

lpszMailboxName

A null-terminated string which contains the name of the mailbox. If the application does not use Unicode, this string is automatically converted to Unicode using the code page specified by the *nCodePage* parameter. This parameter cannot be a NULL pointer or an empty string.

lpszEncodedName

A null-terminated string buffer which will contain the UTF-7 encoded mailbox name. This parameter cannot be NULL and must be large enough to contain the entire name. The string will be null-terminated and will only contain printable ASCII characters. An alternate form of the method accepts a **CString** object which will contain the decoded mailbox name when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the *lpszEncodedName* string buffer. This value must be greater than zero and the buffer must be large enough to store the entire name.

nCodePage

An optional integer value which specifies the code page used when decoding the mailbox name. If the application does not use Unicode, this parameter can be any valid code page. The default value is CP_ACP which specifies the current active code page for the locale. Applications which are built using Unicode should always use the default value.

Return Value

If the method succeeds, it returns the number of characters in the encoded mailbox name. If an error occurs, the method returns zero. To get extended error information, call **GetLastError**.

Remarks

Mailbox names which contain non-ASCII characters must be converted to a modified version of UTF-7 encoding as defined by the IMAP4 protocol. This encoding allows applications to reference mailbox names which contain Unicode characters and provides support for languages other than English. This method is used to convert a mailbox name which contains non-ASCII characters to the UTF-7 encoded required by IMAP servers.

For example, if the *lpszMailboxName* parameter specifies the string **"Δοκιμή"**, this method will encode it as **"&A5QDvwO6A7kDvAOu-"** and return that value in the *lpszEncodedName* buffer. It is

recommended you compile the application using the Unicode character set whenever possible to ensure the mailbox name is encoded correctly.

If the application is not compiled using Unicode, the mailbox name will be automatically converted to Unicode using the code page specified by the ***nCodePage*** parameter. If the code page does not support the characters used in the mailbox name, this can result in an incorrect conversion. Using the previous example, the mailbox name will always encode correctly if the Unicode version of the method is used, but the ANSI version of the method will only encode it correctly if the code page is specified as 1253 (Greek).

This method should not be used to convert strings between Unicode and UTF-7. The encoding format is specifically intended for use with the IMAP4 protocol and should never be used for general purpose encoding or decoding. If you need to convert a Unicode string to UTF-7, use the Win32 API function **WideCharToMultiByte**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, DecodeMailboxName, GetFirstMailbox, GetNextMailbox, RenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::EnumMessages Method

```
INT EnumMessages(
   UINT nFirstMessageId,
   UINT nLastMessageId,
   DWORD dwMessageFlags,
   LPIMAPMESSAGE lpMessageList,
   INT nMaxMessages
);

INT EnumMessages(
   LPIMAPMESSAGE lpMessageList,
   INT nMaxMessages
);
```

The **EnumMessages** method enumerates the messages in the current mailbox, populating an array of IMAPMESSAGE structures which contain information about each message.

Parameters

nFirstMessageId

An unsigned integer value which specifies the first message to enumerate. This value must be greater than zero and specify a valid message identifier. The first message in the mailbox has a value of one.

nLastMessageId

An unsigned integer value which specifies the last message to enumerate. This value must be greater than or equal to the value of the ***nFirstMessageId*** parameter. A special value of 0xFFFFFFFF can be used to specify the last message in the mailbox.

dwMessageFlags

This parameter is used to determine which messages are enumerated. If the value is zero, then all applicable messages will be enumerated. If the value is non-zero, only those messages which have at least one of the specified flags will be returned. More than one flag can be specified by using a bitwise operator.

Value	Description
IMAP_FLAG_ANSWERED	Return only those messages which have been answered.
IMAP_FLAG_DELETED	Return only those messages which have been marked for deletion.
IMAP_FLAG_DRAFT	Return only those messages which have been marked as draft copies.
IMAP_FLAG_URGENT	Return only those messages which have been flagged for urgent or special attention.
IMAP_FLAG_RECENT	Return only those messages which have been recently added to the mailbox.
IMAP_FLAG_SEEN	Return only those messages which have been read.

dwReserved

A reserved parameter. This value should always be set to zero.

lpMessageList

A pointer to an array of IMAPMESSAGE structures which will contain information about each of the messages returned by the server. This parameter cannot be NULL.

nMaxMessages

An integer value which specifies the maximum size of the IMAPMESSAGE array that was passed to the function. This value must be at least one.

Return Value

If the method succeeds, the return value is the number of messages that were enumerated. If no messages match the specified criteria, the method will return a value of zero. If an error is encountered, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the message UID is being stored locally by the client to identify the message over multiple sessions, it must also store the mailbox UID. Only the combination of the mailbox name, mailbox UID and message UID can be used to uniquely identify a given message on the server. Although IMAP server implementations are encouraged to maintain persistent message UIDs, they are not required to do so and those values may change if the mailbox UID changes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetMessage, GetMessageCount, GetMessageFlags, GetMessageHeaders, GetMessageSize, GetMessageUID, IMAPMESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::EnumMessageUids Method

```
INT EnumMessageUids(
   UINT nFirstMessageId,
   UINT nLastMessageId,
   DWORD dwMessageFlags,
   UINT *lpSequenceIdList,
   UINT *lpMessageUidList
   INT nMaxMessages
);

INT EnumMessageUids(
   UINT *lpSequenceIdList,
   DWORD *lpMessageUidList
   INT nMaxMessages
);
```

The **EnumMessageUids** method populates an array of message sequence IDs and their corresponding message UIDs.

Parameters

nFirstMessageId

An unsigned integer value which specifies the first message to enumerate. This value must be greater than zero and specify a valid message identifier. The first message in the mailbox has a value of one.

nLastMessageId

An unsigned integer value which specifies the last message to enumerate. This value must be greater than or equal to the value of the ***nFirstMessageId*** parameter. A special value of 0xFFFFFFFF (-1) can be used to specify the last message in the mailbox.

dwMessageFlags

This parameter is used to determine which messages are enumerated. If the value is zero, then all messages in the mailbox will be enumerated. If the value is non-zero, only those messages which have at least one of the specified flags will be returned. The following flags can be used to filter which messages are included in the result:

Value	Description
IMAP_FLAG_ANSWERED	Return only those messages which have been answered.
IMAP_FLAG_DELETED	Return only those messages which have been marked for deletion.
IMAP_FLAG_DRAFT	Return only those messages which have been marked as draft copies.
IMAP_FLAG_URGENT	Return only those messages which have been flagged for urgent or special attention.
IMAP_FLAG_RECENT	Return only those messages which have been recently added to the mailbox.
IMAP_FLAG_SEEN	Return only those messages which have been read.

lpSequenceIdList

A pointer to an array of unsigned integers that will receive the message sequence numbers for each enumerated message. The message sequence numbers are what most methods use to specify a message in the current mailbox. These values should never be stored by your application. This parameter must not be NULL.

lpMessageUidList

A pointer to an array of unsigned integers that will receive the message UIDs corresponding to the messages in the sequence. The message UIDs are unique values which persist across multiple client sessions. These values can be stored by your application and then mapped to their corresponding sequential message ID. This parameter must not be NULL.

nMaxMessages

An integer value which specifies the maximum size of both arrays. This value cannot be zero.

Return Value

If the method succeeds, the return value is the number of entities populated in both the *lpSequenceIdList* and *lpMessageUidList* arrays. If no messages match the specified criteria, the method will return a value of zero. If an error is encountered, the method returns IMAP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **EnumMessageUids** method retrieves a list of messages in the specified range and populates parallel arrays with each message's sequence number and corresponding unique identifier (UID). This enables applications to correlate transient message numbers with persistent UIDs for more reliable tracking across sessions. The *lpSequenceIdList* and *lpMessageUidList* arrays must be the same size, and both must be large enough to store the number of values specified by the *nMaxMessages* parameter. Both the sequential and unique message IDs are unsigned integer values, but they cannot be used interchangeably. Note that methods such as **GetMessage** and **DeleteMessage** expect sequential message numbers, not message UIDs.

If your application stores the message UIDs to maintain some persistent information about the mailbox, this method will enable you to lookup those message UIDs and find the corresponding sequential message number. In most cases, you would want to call this method immediately after calling **SelectMailbox**. This will cause the client to request a list of message UIDs in the selected mailbox within the range that you specify. Mailboxes with a very large number of messages may cause the application to block for an extended period of time while the list is returned and processed.

To map all messages in the selected mailbox, set `nFirstMessageId = 1`, `nLastMessageId = (UINT)-1` and `dwMessageFlags = 0`.

If the message UID is being stored locally by the client to identify the message over multiple sessions, it must also store the mailbox UID. Only the combination of the mailbox name, mailbox UID and message UID can be used to uniquely identify a given message on the server. Although IMAP server implementations are encouraged to maintain persistent message UIDs, they are not required to do so. Message UIDs may change if the mailbox is recreated or its unique identifier (UIDVALIDITY) is reset.

In some cases, particularly with newly created or infrequently used mailboxes, the message sequence numbers may *coincidentally* match the UIDs assigned by the server. For example, the first message in a new mailbox might have both a message sequence number and UID of 1. However, this is not guaranteed, and applications should never assume that sequence numbers and UIDs are interchangeable. Sequence numbers are dynamic and reflect the current ordering of messages in the selected mailbox. They can change between sessions or when messages are added or deleted. UIDs, on the other hand, are intended to be persistent identifiers that remain associated with a specific

message, even if its sequence number changes.

It is important to note that a message UID is not the same as the Message-ID header field found within the message content. The UID is a unique numeric identifier assigned by the IMAP server to a message in a specific mailbox. It is used by the protocol to track messages even as their sequence numbers change due to mailbox activity (e.g.: new messages arriving or messages being deleted). In contrast, the Message-ID header is a string value generated by the sending email client when the message is created. It is part of the message headers and intended to be globally unique for email threading and deduplication purposes, but it has no direct relationship to the UID used by the server.

Example

This example demonstrates how to enumerate all messages in the currently selected mailbox and retrieve the mapping between their sequential message numbers and server-assigned UIDs. This is useful when your application needs to track messages persistently across sessions using UIDs, while still interacting with the API methods that require sequence numbers such as **GetMessage**.

```
// Get some information about the currently selected mailbox
UINT nLastMessage = 0;
DWORD dwMailboxSize = 0;
INT nMessageCount = imapClient.GetMessageCount(&nLastMessage, &dwMailboxSize);

_tprintf(_T("There are %d messages in %s (last message is %u, mailbox size is %lu
bytes)\n"),
         nMessageCount, szMailbox, nLastMessage, dwMailboxSize);

if (nMessageCount < 1)
{
    // There are no messages in the current mailbox
    return;
}

// Allocate memory for the array of sequence numbers and UIDs
UINT *pSeqList = new UINT[nMessageCount];
UINT *pUidList = new UINT[nMessageCount];

// Populate the arrays with message sequence numbers and corresponding UIDs
INT nUidCount = imapClient.EnumMessageUids(pSeqList, pUidList, nMessageCount);

if (nUidCount > 0 && nUidCount <= nMessageCount)
{
    // Display the mapping of message sequence numbers to UIDs
    for (INT nIndex = 0; nIndex < nUidCount; nIndex++)
    {
        _tprintf(_T("\tMessage %u was assigned UID %u\n"), pSeqList[nIndex],
pUidList[nIndex]);
    }
}
else
{
    // Enumeration failed or no matching messages
    DWORD dwError = imapClient.GetLastError();
    _tprintf(_T("Unable to enumerate message UIDs (error code: 0x%lx)\n"),
dwError);
}

// Free the memory allocated for our arrays
delete[] pSeqList;
delete[] pUidList;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

EnumMessages, GetMessage, GetMessageCount, GetMessageUid, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::ExamineMailbox Method

```
INT ExamineMailbox(
   LPCTSTR lpszMailbox,
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ExamineMailbox** method selects the specified mailbox for read-only access.

Parameters

lpszMailbox

A null-terminated string which specifies the new mailbox to be examined.

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ExamineMailbox** method is used to select a mailbox in read-only mode. Messages can be read, but they cannot be modified or deleted from the mailbox and new messages will not lose their status as new messages if they are accessed.

If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to examining the new mailbox.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **ExamineMailbox** again with the same mailbox name.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-write mode, use the **SelectMailbox** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMailbox, GetFirstMailbox, GetMailboxStatus, GetNextMailbox, RenameMailbox, ReselectMailbox, SelectMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::ExpungeMailbox Method

```
INT ExpungeMailbox(
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ExpungeMailbox** method removes all messages marked for deletion from the current mailbox.

Parameters

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ExpungeMailbox** method causes all messages marked for deletion to be removed from the mailbox. Note that this can cause the mailbox's UID to change, and potentially invalidate the current message UIDs. It is recommended that applications use the information returned in the IMAPMAILBOX structure to update any internal state information stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, ExamineMailbox, GetFirstMailbox, GetNextMailbox, RenameMailbox, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetCapability Method

```
INT GetCapability(
   LPTSTR lpszCapability,
   INT nMaxLength
);

INT GetCapability(
   CString& strCapability
);
```

The **GetCapability** method returns a string which identifies the capabilities of the IMAP server.

Parameters

lpszCapability

A null-terminated string buffer that will contain one or more tokens separated by spaces which identify the capabilities of the IMAP server.

nMaxLength

The maximum length of the capability string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the capability string, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns a string that contains one or more tokens separated by whitespace. Each token identifies a capability of the server. The following table lists some of the common capabilities:

Capability	Description
ACL	The RFC 2086 ACL extension
BINARY	The RFC 3516 binary content extension
CHILDREN	The RFC 3348 child mailbox extension
ID	The RFC 2971 ID extension
IDLE	The RFC 2177 IDLE extension
LOGINDISABLED	The RFC 2595 TLS/SSL extension
LOGINREFERRALS	The RFC 2221 login referrals extension
MAILBOXREFERRALS	The RFC 2193 mailbox referrals extension
MULTIAPPEND	The RFC 3501 MULTIAPPEND extension
NAMESPACE	The RFC 2342 namespace Extension
QUOTA	The RFC 2087 QUOTA extension
STARTTLS	The RFC 2595 TLS/SSL extension
UNSELECT	The RFC 3691 UNSELECT extension

Additional capabilities may be supported by your server. Note that experimental or custom capabilities

are always prefixed with the letter X. A list of standard IMAP capabilities is maintained by the Internet Assigned Numbers Authority (IANA) at www.iana.org.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetCurrentMailbox Method

```
INT GetCurrentMailbox(
   LPTSTR lpszMailbox,
   INT nMaxLength
);

INT GetCurrentMailbox(
   CString& strMailbox
);
```

The **GetCurrentMailbox** method returns the name of the current mailbox that has been selected.

Parameters

lpszMailbox

A null-terminated string buffer that will contain the current mailbox name. If the application uses the Unicode character set, the mailbox name will automatically be converted from modified UTF-7 encoding to Unicode. If the application does not use Unicode, the mailbox name will be returned using modified UTF-7 encoding. An alternate form of the method accepts a **CString** object which will contain the mailbox name when the method returns.

nMaxLength

The maximum length of the mailbox string, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the current mailbox name, not including the terminating null character. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

It is recommended you compile the application using the Unicode character set, particularly if it is possible that mailbox names will contain Unicode characters or symbols. This will ensure that mailbox names are automatically converted from modified UTF-7 to UTF-16 Unicode. If Unicode is not used and a mailbox name contains non-ASCII characters, the name will be returned using a modified UTF-7 encoding format as defined by the IMAP4 standard. The mailbox name will not be returned as UTF-8 Unicode or as ANSI text converted to the current locale.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ExamineMailbox, GetFirstMailbox, GetNextMailbox, SelectMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetDeletedMessages Method

```
INT GetDeletedMessages(
   UINT *lpnMessageIds,
   INT nMaxMessages
);
```

The **GetDeletedMessages** method returns the message identifiers for those messages that have been marked for deletion in the current mailbox.

Parameters

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which have been marked for deletion in the current mailbox. This parameter may be NULL, in which case the method will return the number of deleted messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of messages marked for deletion in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected.

To remove the deleted flag from a message, use the **UndeleteMessage** method. To prevent all messages in the current mailbox from being expunged, use the **ReselectMailbox** method to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetMessageCount, GetMessageFlags, GetNewMessages, GetUnseenMessages, SearchMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetFirstMailbox Method

```
INT GetFirstMailbox(
   LPCTSTR lpszReference,
   LPCTSTR lpszWildcard,
   DWORD dwOptions,
   LPTSTR lpszMailbox,
   INT nMaxLength
   LPDWORD lpdwFlags
);

INT GetFirstMailbox(
   CString& strMailbox,
   LPDWORD lpdwFlags
);
```

The **GetFirstMailbox** method returns the name of the first matching mailbox.

Parameters

lpszReference

A null-terminated string which specifies the reference name. An empty string or NULL pointer specifies that the default mailbox hierarchy for the current user is returned. If the reference name is provided, this must be the name of a mailbox or a level of the mailbox hierarchy which provides the context in which the mailbox name is interpreted.

lpszWildcard

A null-terminated string which specifies the mailbox name to match. The wildcard character "*" may be used to match any portion of the mailbox hierarchy, including the delimiter. The wildcard character "%" matches any portion of the mailbox name, but does not match the mailbox delimiter. An empty string or NULL pointer specifies that all mailboxes in the context of the *lpszReference* parameter should be returned.

dwOptions

Specifies one or more options which controls how mailboxes are returned by the method. The options are bit flags which may be combined using a bitwise operator. One or more of the following values may be used:

Value	Description
IMAP_LIST_DEFAULT	This option specifies that all regular, selectable mailboxes should be returned.
IMAP_LIST_SUBSCRIBED	This option specifies that only subscribed mailboxes should be returned.
IMAP_LIST_FOLDERS	This option specifies that non-selectable mailbox folders should also be returned.
IMAP_LIST_HIDDEN	This option specifies that hidden mailboxes should be returned.
IMAP_LIST_INFERIOR	This option specifies that inferior mailboxes should be returned if an explicit wildcard mask is not specified.

lpszMailbox

A null-terminated string buffer which will contain the first matching mailbox. This parameter cannot

be NULL. A minimum buffer size of at least 128 character is recommended. If the application uses the Unicode character set, the mailbox name will automatically be converted from modified UTF-7 encoding to Unicode. If the application does not use Unicode, the mailbox name will be returned using modified UTF-7 encoding.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the first matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Value	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the method succeeds, it returns the length of the mailbox name. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstMailbox** method is used to begin enumerating the available mailboxes for the current user on the IMAP server. Subsequent mailbox names are returned by calling **GetNextMailbox** until the method returns IMAP_ERROR with an error code of ST_ERROR_NO_MORE_MAILBOXES.

It is recommended you compile the application using the Unicode character set, particularly if it is possible that mailbox names will contain Unicode characters or symbols. This will ensure that mailbox names are automatically converted from modified UTF-7 to UTF-16 Unicode. If Unicode is not used and a mailbox name contains non-ASCII characters, the name will be returned using a modified UTF-7 encoding format as defined by the IMAP4 standard. The mailbox name will not be returned as UTF-8 Unicode or as ANSI text converted to the current locale.

The method of the ***lpszReference*** and ***lpszWildcard*** parameters are implementation dependent and generally are tied to the underlying operating system. On a UNIX based system, it can be helpful to

think of the reference name as the directory where mailbox folders are stored, and the mailbox name as the name to search for in that directory and any subdirectories, if applicable. If the reference name is an empty string or NULL pointer, this typically refers to the current user's home directory.

Generally speaking, a reference name should only be specified if you or the user of the application knows the directory structure on the IMAP server. Incorrectly using a reference name can have serious negative side-effects. For example, specifying a reference name of "/" on a UNIX based system could cause the IMAP server to return search every directory on the system for a matching mailbox name. Similarly, the IMAP server may be unable to distinguish between regular files in the user's home directory and mailboxes. For this reason, most IMAP clients require that the user specify the directory on the server where their mailboxes are stored. Typically this is subdirectory named "mail" or "Mail" under the user's home directory. For non-UNIX servers, the mailbox hierarchy may represented differently, including a flat hierarchy.

Hidden mailboxes are those mailboxes which use the UNIX convention of the name beginning with a period. Therefore, a mailbox named ".secrets" would not normally be returned by the **GetFirstMailbox** and **GetNextMailbox** methods. The IMAP_LIST_HIDDEN option causes all mailboxes to be returned.

The IMAP_LIST_INFERIOR option will return inferior mailboxes (mailboxes located in folders or subdirectories) if a wildcard mask is not specified. If a wildcard mask is specified, this option has no effect and only those mailboxes which match the wildcard will be returned.

Subscribed mailboxes are those which were specified using the **SubscribeMailbox** method. Marked mailboxes are typically those which have some special importance to the user.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMailbox, GetCurrentMailbox, GetMailboxStatus, GetNextMailbox, RenameMailbox, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetHeaderValue Method

```
INT GetHeaderValue(
   UINT nMessageId,
   UINT nMessagePart,
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxLength
);

INT GetHeaderValue(
   UINT nMessageId,
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxLength
);

INT GetHeaderValue(
   UINT nMessageId,
   UINT nMessagePart,
   LPCTSTR lpszHeader,
   CString& strValue
);

INT GetHeaderValue(
   UINT nMessageId,
   LPCTSTR lpszHeader,
   CString& strValue
);
```

The **GetHeaderValue** method returns the value of a header field in the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that the header value will be be retrieved from. Message parts start with a value of one. A value of zero specifies that the RFC822 header field for the message will be used.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the header field value. If the header field is not present in the message, the method will return a value of zero. If the method fails, the return value is

IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetHeaderValue** method returns the value of header field from the specified message. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessageHeaders, GetMessageId, GetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetIdleThreadId Method

```
DWORD GetIdleThreadId();
```

The **GetIdleThreadId** method returns ID of the thread that is monitoring the client session.

Parameters

None.

Return Value

If the method succeeds, it returns the ID of the thread that is checking for update notifications from the server. If there is no active thread monitoring the client session, the method will return zero.

Remarks

The worker thread that monitors the client connection in the background can terminate if an IMAP command is sent to the server, if the **Cancel** method is called or if the client disconnects from the server. The **GetIdleThreadId** method enables the application to determine if this background thread is still active or not.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Idle, ImapIdleProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMailboxDelimiter Method

```
INT GetMailboxDelimiter(
   LPCTSTR lpszMailbox,
   LPTSTR lpszDelimiter,
   INT nMaxLength
);

INT GetMailboxDelimiter(
   LPCTSTR lpszMailbox,
   CString& strDelimiter
);
```

The **GetMailboxDelimiter** method returns the hierarchical path delimiter used for the specified mailbox.

Parameters

lpszMailbox

A null-terminated string which specifies the mailbox name. This parameter may be NULL or an empty string, in which case the default delimiter will be returned.

lpszDelimiter

A null-terminated string buffer that will contain the mailbox delimiter. An alternate form of this method accepts a **CString** object that will contain the delimiter when the method returns.

nMaxLength

The maximum length of the delimiter string, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the delimiter for the specified mailbox, not including the terminating null character. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the IMAP server supports multiple levels of mailboxes, then a special character or sequence of characters are used as delimiters between different levels of the mailbox hierarchy. On most systems, including most UNIX and Windows platforms, the delimiter is the forward slash "/" character.

It is possible that an IMAP server may only support a flat namespace, in which case this method will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ExamineMailbox, SelectMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMailboxSize Method

```
DWORD GetMailboxSize(
   LPCTSTR lpszMailbox
);
```

The **GetMailboxSize** method returns the size of the specified mailbox.

Parameters

lpszMailbox

A null-terminated string which specifies the mailbox name.

Return Value

If the method succeeds, it returns the size of the mailbox. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMailboxSize** method may require a significant amount of time to calculate the mailbox size if there are a large number of messages in the mailbox. If the specified mailbox is not currently selected, then the current mailbox is unselected, the new mailbox is selected and the size calculated, and then the original mailbox is re-selected. This will have the side-effect of causing any messages marked for deletion to be expunged from the mailbox.

Because it can potentially result in long delays, it is not recommended that an application calculate the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, GetFirstMailbox, GetNextMailbox, RenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMailboxStatus Method

```
INT GetMailboxStatus(
   LPCTSTR lpszMailbox,
   LPIMAPMAILBOXSTATUS lpMailboxStatus
);
```

The **GetMailboxStatus** method returns status information about the specified mailbox.

Parameters

lpszMailbox

A null-terminated string that specifies the mailbox to return information about.

lpMailboxStatus

A pointer to an IMAPMAILBOXSTATUS structure which contains status information about the specified mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMailboxStatus** method enables an application to obtain status information about a mailbox without having to select another mailbox or open a second connection to the IMAP server to examine the mailbox. The information returned is a subset of the information returned when a mailbox is selected.

If you need to check the status of a mailbox which contains Unicode characters or symbols, it is recommended you use the Unicode character set when compiling your application. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. If you do not use Unicode, this method will only accept mailbox names using modified UTF-7 encoding or UTF-8 encoding.

Note that obtaining status information for a mailbox may be a slow operation. It may require that server open the mailbox in read-only mode internally in order to obtain some of the status information. For this reason, this method should not be used to check for new messages; use the **CheckMailbox** method instead.

Some IMAP servers may return an error if you attempt to obtain status information about the currently selected mailbox. The protocol standard states that clients should not use this method on the currently selected mailbox, and should instead use the information returned by the **SelectMailbox** or **ExamineMailbox** methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CheckMailbox, ExamineMailbox, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMailboxUID Method

```
DWORD GetMailboxUID(
   LPCTSTR lpszMailbox
);
```

The **GetMailboxUID** method returns the unique identifier for the specified mailbox.

Parameters

lpszMailbox

A null-terminated string that specifies the mailbox name.

Return Value

If the method succeeds, it returns a non-zero value. If no unique identifier is assigned to the mailbox, the method will return zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMailboxUID** method returns an unsigned 32-bit value which uniquely identifies the mailbox and corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

An application can use the mailbox UID value in combination with the message UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCurrentMailbox, GetMailboxStatus, GetMessageUID

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessage Method

```
INT GetMessage(
   UINT nMessageId,
   UINT nMessagePart,
   LPBYTE lpMessage,
   LPDWORD lpdwLength,
   DWORD dwOptions
);

INT GetMessage(
   UINT nMessageId,
   UINT nMessagePart,
   HGLOBAL *lpMessage,
   LPDWORD lpdwLength,
   DWORD dwOptions
);

INT GetMessage(
   UINT nMessageId,
   UINT nMessagePart,
   CString& strMessage,
   DWORD dwOptions
);

INT GetMessage(
   UINT nMessageId,
   CString& strMessage
);
```

The **GetMessage** method retrieves a message from the server.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

lpMessage

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpMessage*** parameter. If the ***lpMessage*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description

IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.
IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessage** method is used to retrieve a message from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the *lpMessage* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpMessage* parameter point to a global memory handle which will contain the message data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine if a message is a multipart MIME message, use the **GetMessageParts** method. The return value specifies the number of parts in the message, with a value greater than one indicating that it is a multipart message. Combining the IMAP_SECTION_HEADER and IMAP_SECTION_BODY options will only return the header and body for the specified message if the ***nMessagePart*** parameter is zero. Due to a limitation of the IMAP FETCH command, if a message part is specified then only the body of that message part will be returned.

Note that unlike the SocketTools MIME API which considers the first message part to be zero, the IMAP protocol defines the first message part to be one.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetMessageCount, GetMessageHeaders, GetMessageParts, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageCount Method

```
INT GetMessageCount(
   UINT *lpnLastMessage,
   DWORD *lpdwMailboxSize
);
```

The **GetMessageCount** method returns the number of messages that are available in the currently selected mailbox, and optionally the size of the mailbox in bytes.

Parameters

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. If a NULL value is specified, this argument is ignored.

Return Value

If the method succeeds, it returns the number of messages that are currently available. If no messages are available, this method will return zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageCount** method is provided for compatibility with the POP3 class. The *lpnLastMessage* parameter will always contain the same value returned by the method since there is no distinction between the message count and the last available message. This is because messages that are marked for deletion on an IMAP server can still be accessed until the mailbox is expunged or unselected. This differs from the POP3 protocol, where messages cannot be accessed once they have been marked for deletion.

If the *lpdwMailboxSize* parameter is specified, this method will call **GetMailboxSize** to determine the size of the currently selected mailbox. Unlike the POP3 protocol, calculating the mailbox size may require a significant amount of time if there are a large number of messages in the mailbox. It is not recommended that an application request the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMessage, GetHeaderValue, GetMailboxSize, GetMessage, GetMessageHeaders, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageFlags Method

```
BOOL GetMessageFlags(
   UINT nMessageId,
   LPDWORD lpdwMessageFlags
);
```

The **GetMessageFlags** method returns the message flags for the specified message.

Parameters

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

lpdwMessageFlags

Pointer to an unsigned integer that will contain the message flags for the specified message. The value may be zero, or one or more of the following values:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the method succeeds, the return value is non-zero and the ***lpdwMessageFlags*** parameter contains the status flags for the specified message. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

DeleteMessage, GetMessageCount, SetMessageFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageHeaders Method

```
INT GetMessageHeaders(
   UINT nMessageId,
   LPBYTE lpHeaders,
   LPDWORD lpdwLength
);

INT GetMessageHeaders(
   UINT nMessageId,
   HGLOBAL *lpHeaders,
   LPDWORD lpdwLength
);

INT GetMessageHeaders(
   UINT nMessageId,
   CString& strHeaders
);
```

The **GetMessageHeaders** method retrieves the headers for the specified message from the server.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

lpHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpHeaders*** parameter. If the ***lpHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessagHeaders** method is used to retrieve a message header block from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpHeaders*** parameter point to a global memory handle which will contain the file data when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically

fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetMessage, GetMessageCount, GetMessageParts, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageId Method

```
INT GetMessageId(
   UINT nMessageId,
   LPTSTR lpszMessageId,
   INT nMaxLength
);

INT GetMessageId(
   UINT nMessageId,
   CString& strMessageId
);
```

The **GetMessageId** method returns the message identifier string for the specified message.

Parameters

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the length of the unique identifier string. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageId** method returns the message identifier from the Message-ID header of the specified message. The returned value is a string which can be used to identify a specific message, regardless if the message is moved to a different mailbox.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeaderValue, GetMessage, GetMessageHeaders, GetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageParts Method

```
INT GetMessageParts(
   UINT nMessageId
);
```

The **GetMessageParts** method returns the number of parts in a MIME multipart message on the server.

Parameters

nMessageId

Number of message to retrieve the part count for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is the number of parts in the specified message. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageParts** method can be used to determine if a message on the server contains multiple parts. A multipart MIME message typically contains file attachments or multiple representations of the message, such as a version of the message in plain text and another using HTML markup.

If the method returns a value of one, then the message does not contain multiple parts and is a standard RFC822 formatted message. A value greater than one indicates that the message does have multiple parts. The **GetMessageEx** method may be used to retrieve the data for a specific part of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetHeaderValue, GetMessage, GetMessageHeaders, GetMessageId, GetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageSender Method

```
INT GetMessageSender(
   UINT nMessageId,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetMessageSender(
   UINT nMessageId,
   CString& strAddress
);
```

The **GetMessageSender** method returns the sender's address for the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the address. If the sender cannot be determined, the method will return a value of zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageSender** method returns the email address specified in the Return-Path header field. This allows an application to be able to easily determine the sender, without parsing the header or downloading the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeaderValue, GetMessageHeaders, GetMessageId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageSize Method

```
DWORD GetMessageSize(
   UINT nMessageId
);
```

The **GetMessageSize** method returns the size of the specified message.

Parameters

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is the size of the specified message in bytes. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetHeaderValue, GetMessageHeaders, GetMessageId, GetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetMessageUID Method

```
DWORD GetMessageUID(
   UINT nMessageId
);
```

The **GetMessageUID** method returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

nMessageId

Number of message to obtain the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the method will return zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageUID** method returns an unsigned integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque.

An application can use the message UID value in combination with the mailbox UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetCurrentMailbox, GetMailboxStatus, GetMailboxUID

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetNewMessages Method

```
INT GetNewMessages(
   UINT *lpnMessageIds,
   INT nMaxMessages
);
```

The **GetNewMessages** method returns the message identifiers for those messages that have recently been added to the mailbox and have not been read.

Parameters

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have recently been added to the mailbox and have not been read. This parameter may be NULL, in which case the method will return the number of new messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of new messages in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **GetMessage** or **StoreMessage**, it is no longer considered to be a new message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetDeletedMessages, GetMessageCount, GetMessageFlags, GetUnseenMessages, SearchMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetNextMailbox Method

```
INT GetNextMailbox(
   LPTSTR lpszMailbox,
   INT nMaxLength
   LPDWORD lpdwFlags
);

INT GetNextMailbox(
   CString& strMailbox,
   LPDWORD lpdwFlags
);
```

The **GetNextMailbox** method returns the name of the next matching mailbox.

Parameters

lpszMailbox

A null-terminated string buffer which will contain the next matching mailbox. This parameter cannot be NULL. A minimum buffer size of at least 128 character is recommended. If the application uses the Unicode character set, the mailbox name will automatically be converted from modified UTF-7 encoding to Unicode. If the application does not use Unicode, the mailbox name will be returned using modified UTF-7 encoding.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the next matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Value	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the method succeeds, it returns the length of the mailbox name. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetNextMailbox** method returns the next matching mailbox name. When the last mailbox has been returned, the next call to this method will result in an error, with the last error code set to ST_ERROR_NO_MORE_MAILBOXES.

It is recommended you compile the application using the Unicode character set, particularly if it is possible that mailbox names will contain Unicode characters or symbols. This will ensure that mailbox names are automatically converted from modified UTF-7 to UTF-16 Unicode. If Unicode is not used and a mailbox name contains non-ASCII characters, the name will be returned using a modified UTF-7 encoding format as defined by the IMAP4 standard. The mailbox name will not be returned as UTF-8 Unicode or as ANSI text converted to the current locale.

Subscribed mailboxes are those which were specified using the **SubscribeMailbox** method. Marked mailboxes are typically those which have some special importance to the user.

For more information about enumerating the available mailboxes on the IMAP server, refer to the **GetFirstMailbox** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMailbox, GetFirstMailbox, GetMailboxStatus, RenameMailbox, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

Command, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetResultString Method

```
INT GetResultString(
   LPTSTR lpszResult,
   INT cbResult
);

INT GetResultString(
   CString& strResult
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

The buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
IMAP_STATUS_UNUSED	No connection has been established.
IMAP_STATUS_IDLE	The client is current idle and not sending or receiving data.
IMAP_STATUS_CONNECT	The client is establishing a connection with the server.
IMAP_STATUS_READ	The client is reading data from the server.
IMAP_STATUS_WRITE	The client is writing data to the server.
IMAP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

IsBlocking, IsConnected, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetTransferStatus Method

```
INT GetTransferStatus(
   LPIMAPTRANSFERSTATUS lpStatus
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an IMAPTRANSFERSTATUS structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this method will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::GetUnseenMessages Method

```
INT GetUnseenMessages(
   UINT *lpnMessageIds,
   INT nMaxMessages
);
```

The **GetUnseenMessages** method returns the message identifiers for those messages that have not been read.

Parameters

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have not been read. This parameter may be NULL, in which case the method will return the number of unseen messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of unseen messages in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **GetMessage** or **StoreMessage**, it is no longer considered to be an unseen message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetDeletedMessages, GetMessageCount, GetMessageFlags, GetNewMessages, SearchMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Idle Method

```
INT Idle(
   IMAPIDLEPROC lpfnIdleProc,
   DWORD_PTR dwParam
);

INT Idle(
   UINT nTimeout,
   DWORD dwOptions,
   IMAPIDLEPROC lpfnIdleProc,
   DWORD_PTR dwParam
);
```

The **Idle** method enables mailbox status monitoring for the client session, allowing the client to receive notifications from the server whenever a new message arrives or a message is expunged from the currently selected mailbox. This is typically used as an alternative to the client periodically polling the server for status information.

Parameters

nTimeout

Specifies the timeout period in seconds to wait for a notification from the server. This parameter is only used when the IMAP_IDLE_WAIT option has been specified.

dwOptions

Specifies the options which should be used when enabling idle monitoring. The following options are supported:

Value	Description
IMAP_IDLE_NOWAIT	The method should return immediately after idle processing has been enabled. When this option is used, the application may continue to perform other methods while the client session is monitored for status updates sent by the server. The client will continue to monitor status changes until an IMAP command issued or the client disconnects from the server.
IMAP_IDLE_WAIT	The method should wait until the server sends a status update, or until the timeout period is reached. The client will stop monitoring status changes when the method returns. If this option is used in a single-threaded application, normal message processing can be impeded, causing the application to appear non-responsive until the timeout period is reached. It is strongly recommended that single-threaded applications with a user interface specify the IMAP_IDLE_NOWAIT option instead.

lpfnIdleProc

A pointer to an IMAPIDLEPROC callback function that will be invoked whenever the server sends an update notification to the client. This parameter must specify a valid function address and cannot be a NULL pointer.

dwParam

A user-defined value that is passed back to the caller whenever the callback function is invoked. This can be used to provide additional state information to the client. If it is not needed, the caller should

use a value of zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Many IMAP servers support the ability to asynchronously send status updates to the client, rather than have the client periodically poll the server. The client enables this feature by calling **Idle** and providing the address of a callback function that will be invoked whenever the server sends an update notification to the client. Typically these updates inform the client that a new message has arrived or that a message has been expunged from the mailbox.

The **Idle** method can operate in two modes, based on the options specified by the caller. If the option IMAP_IDLE_NOWAIT is specified, the method begins monitoring the client session and returns control immediately to the caller. If the server sends a update notification, the callback function will be invoked with information about the status change. If the option IMAP_IDLE_WAIT is specified, the method will block waiting for the server to send a notification message to the client. The method will return when either a message is received or the timeout period is exceeded.

Sending an IMAP command to the server will cause the client to stop monitoring the session for status changes. To explicitly stop monitoring the session, use the **Cancel** method. To determine if the current client session is being monitored, use the **GetIdleThreadId** method. A non-zero return value indicates that the client session is idle and being monitored.

This method works by sending the IDLE command to the server and starting a worker thread which monitors the connection and looks for untagged responses issued by the server. Callbacks will be invoked for EXISTS, EXPUNGE and RECENT messages. Note that some servers may periodically send untagged OK messages to the client, indicating that the connection is still active; these messages are explicitly ignored. Because the monitoring is performed in a different thread, the callback function is invoked in the context of that thread. Client event notifications are disabled while inside the callback function, and the **Idle** method cannot be used to restart monitoring from within a callback function.

Applications should not perform any operation that takes a significant amount of time or updates the user interface from within the callback function. Instead, use flags or send application defined messages to indicate a change in state. For example, if the server sends a notification that a new email message has arrived, the application should not attempt to read the new message and update the user interface from within the callback function. Instead, it could use the **CWindow::PostMessage** method to send an application-defined message to the UI thread indicating the change in state. The application would have a message handler for that Windows message and update the user interface, indicating that a new message has arrived.

An application should never make an assumption about how a particular server may send update notifications to the client. Servers can be configured to use different intervals at which notifications are sent. For example, a server may send new message notifications immediately, but may periodically notify the client when a message has been expunged. Alternatively, a server may only send notifications at fixed intervals, in which case the client would not be notified of any new messages until the interval period is reached. It is not possible for a client to know what a particular server's update interval is. Applications that require that degree of control should not use **Idle** and should poll the server instead.

Example

```
// Begin monitoring the client session for status changes; when a new
```

```
// message arrives or a message is expunged, the UpdateHandler callback
// function will be invoked.
BOOL CUserApp::MonitorUpdates()
{
    INT nResult;

    nResult = m_imapClient.Idle(UpdateHandler, (DWORD)this);

    if (nResult == IMAP_ERROR)
    {
        m_imapClient.ShowError();
        return FALSE;
    }

    return TRUE;
}

// This function is called whenever the server notifies the client
// that a new message has arrived or a message has been expunged from
// the current mailbox
#define WM_APP_NEWMESSAGE (WM_APP+1)
#define WM_APP_EXPUNGED   (WM_APP+2)

VOID CALLBACK CUserApp::UpdateHandler(HCLIENT hClient, UINT nUpdateId, UINT
nMessageId, DWORD_PTR dwParam)
{
    CUserApp *pThis = (CUserApp *)dwParam;

    switch (nUpdateId)
    {
    case IMAP_UPDATE_MESSAGE:
        {
            // Send a message indicating that a new message has arrived
            pThis->m_pMainWnd->PostMessage(WM_APP_NEWMESSAGE, nMessageId, 0);
        }
        break;

    case IMAP_UPDATE_EXPUNGE:
        {
            // Send a message indicating that a message has been expunged
            pThis->m_pMainWnd->PostMessage(WM_APP_EXPUNGED, nMessageId, 0);
        }
        break;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetIdleThreadId, ImapIdleProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

CImapClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Login Method

```
INT Login(
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);

INT Login(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **Login** method authenticates the specified user in on the server. This method must be called after the connection has been established, and before attempting to retrieve messages or perform any other method on the server.

Parameters

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Value	Description
IMAP_AUTH_LOGIN	Standard cleartext username and password is sent to the server. This authentication method is supported by all servers. Note that some servers may only support LOGIN authentication if a secure connection has been established.
IMAP_AUTH_PLAIN	Login using the PLAIN authentication mechanism as defined in RFC 4959. This authentication method is supported by most servers, although some may require that client establish a secure connection.
IMAP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
IMAP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
IMAP_AUTH_ANONYMOUS	Login using the anonymous Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4505. If this authentication method is specified, the ***lpszUserName*** parameter should specify a name or email address

	acceptable to the mail server. The *lpszPassword* parameter is ignored and may be NULL.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the current client session.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the current client session. If you are using the IMAP_AUTH_XOAUTH2 or IMAP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the bearer token provided by the mail service.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

In some cases, the user may be pre-authenticated by the server. In this case, the method will fail with the last error set to ST_ERROR_ALREADY_AUTHENTICATED. If a particular authentication method is not supported by the server, the last error will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE. For compatibility with the greatest number of servers, it is recommended that you use IMAP_AUTH_LOGIN as the authentication method.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The IMAP_AUTH_XOAUTH2 and IMAP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::OpenMessage Method

```
INT OpenMessage(
   UINT nMessageId,
   UINT nMessagePart,
   DWORD dwOffset,
   LPDWORD lpdwLength,
   DWORD dwOptions
);

INT OpenMessage(
   UINT nMessageId,
   DWORD dwOptions
);
```

The **OpenMessage** method opens a message or a specific part of a multipart message in the current mailbox. The message data may also be limited a specific byte offset and length, which can be useful for previewing the contents.

Parameters

nMessageId

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

dwOffset

The byte offset into the message. This parameter can be used in conjunction with the ***lpdwLength*** parameter to return a specific part of a message. A value of zero specifies the beginning of the message.

lpdwLength

A pointer to an unsigned integer value which should be initialized to the maximum number of bytes to be read, and will contain the size of the message when the method returns. To specify the entire message, from the offset specified by the ***dwOffset*** parameter to the end of the message, initialize the ***lpdwLength*** parameter to a value of -1. This parameter may be NULL if the message size is not needed.

dwOptions

The low order word of this parameter specifies how the message data will be returned. It may be one of the following values:

Value	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.

IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **OpenMessageEx** method uses the FETCH command to access the specified message on the server. The client can then use the **Read** method to read the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseMessage, GetMessage, GetMessageHeaders, OpenMessage, Read, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   IMAPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
IMAP_EVENT_CONNECT	The connection to the server has completed.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
IMAP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
IMAP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
IMAP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **ImapEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **ImapEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a message is being retrieved. The IMAP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one IMAP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, ImapEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::RenameMailbox Method

```
INT RenameMailbox(
   LPCTSTR lpszOldMailbox,
   LPCTSTR lpszNewMailbox
);
```

The **RenameMailbox** method renames an existing mailbox.

Parameters

lpszOldMailbox

A null-terminated string which specifies the mailbox to be renamed.

lpszNewMailbox

A null-terminated string which specifies the new mailbox name.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RenameMailbox** method renames an existing mailbox on the server. The new mailbox name cannot exist on the server, or the method will fail.

If the existing mailbox name contains inferior hierarchical names (mailboxes under the specified mailbox) then those mailboxes will also be renamed. For example, if the mailbox "Mail/Pictures" contains two mailboxes, "Personal" and "Work" and it is renamed to "Mail/Images" then the two mailboxes under it would be automatically renamed to "Mail/Images/Personal" and "Mail/Images/Work".

If the mailbox being renamed is the currently selected mailbox, the current mailbox will be unselected and any messages marked for deletion will be expunged. The new mailbox name will then automatically be re-selected. To prevent deleted messages from being removed from the mailbox prior to being renamed, use the **UnselectMailbox** method to unselect the current mailbox before calling **RenameMailbox**. Note that if the rename operation fails, the client may be left in an unselected state.

It is permitted to rename the special mailbox INBOX. In this case, the messages will be moved from the INBOX mailbox to the new mailbox. If the INBOX mailbox is currently selected, the new mailbox will not automatically be selected. INBOX will remain the selected mailbox.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateMailbox, DeleteMailbox, GetFirstMailbox, GetNextMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::ReselectMailbox Method

```
BOOL ReselectMailbox(
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ReselectMailbox** method reselects the current mailbox and returns updated information about the status of the mailbox.

Parameters

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains updated information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ReselectMailbox** method forces the current mailbox to be reselected and returns updated information about the status of the mailbox. Deleted messages are not expunged from the mailbox and remain marked for deletion.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ExamineMailbox, SelectMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::SearchMailbox Method

```
LONG SearchMailbox(
   LPCTSTR lpszCriteria,
   LPCTSTR lpszCharacterSet,
   UINT *lpnMessageIds,
   LONG nMaxMessages
);

LONG SearchMailbox(
   LPCTSTR lpszCriteria,
   UINT *lpnMessageIds,
   LONG nMaxMessages
);
```

The **SearchMailbox** method searches the current mailbox for messages that match the specified criteria, returning matching message identifiers.

Parameters

lpszCriteria

A null-terminated string which specifies the search criteria.

lpszCharacterSet

A null-terminated string which specifies the character set to use when searching the mailbox. If this parameter is NULL or an empty string, the default US-ASCII character set will be used.

dwReserved

A reserved parameter which should be set to the value 0.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which match the search criteria in the current mailbox. This parameter may be NULL, in which case the method will return the number of matching messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of messages that meet the search criteria in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SearchMailbox** method is used to search a mailbox for messages which match a given criteria and return a list of the matching message identifiers. The search criteria is composed of one or more search keywords and and optional value to match against. String searches are not case sensitive and partial matches in the message are returned.

The following search keywords are recognized:

Keyword	Description
ANSWERED	Match those messages which have the IMAP_FLAG_ANSWERED flag set.

BCC *address*	Match those messages which contain the specified address in the BCC header field.
BEFORE *date*	Match those messages which were added to the mailbox prior to the specified date.
BODY *string*	Match those messages where the body contains the specified string.
CC *address*	Match those messages which contain the specified address in the CC header field.
DELETED	Match those messages which have the IMAP_FLAG_DELETED flag set.
DRAFT	Match those messages which have the IMAP_FLAG_DRAFT flag set.
FLAGGED	Match those messages which have the IMAP_FLAG_URGENT flag set.
FROM *address*	Match those messages which contain the specified address in the FROM header field.
HEADER *field string*	Match those messages which contain the string in the specified header field. If no string is specified, then all messages which contain the header will be matched.
LARGER *size*	Match those messages which are larger than the specified size in bytes.
NEW	Match those messages which have the IMAP_FLAG_RECENT flag set, but not the IMAP_FLAG_SEEN flag.
OLD	Match those messages which do not have the IMAP_FLAG_RECENT flag set.
ON *date*	Match those messages which were added on the specified date.
RECENT	Match those messages which have the IMAP_FLAG_RECENT flag set.
SEEN	Match those messages which have the IMAP_FLAG_SEEN flag set.
SENTBEFORE *date*	Match those messages whose Date header value is earlier than the specified date.
SENTON *date*	Match those messages whose Date header value is the same as the specified date.
SENTSINCE *date*	Match those messages whose Date header value is later than the specified date.
SINCE *date*	Match those messages added to the mailbox after the specified date.
SMALLER *size*	Match those messages which are smaller than the specified size in bytes.
SUBJECT *string*	Match those messages whose Subject header contains the specified string.
TEXT *string*	Match those messages whose headers or body contains the specified string.
TO *address*	Match those messages which contain the specified address in the TO header field.
UID *sequence*	Match those messages with unique identifiers in the sequence set.
UNANSWERED	Match those messages which do not have the IMAP_FLAG_ANSWERED

	flag set.
UNDELETED	Match those messages which do not have the IMAP_FLAG_DELETED flag set.
UNDRAFT	Match those messages which do not have the IMAP_FLAG_DRAFT flag set.
UNFLAGGED	Match those messages which do not have the IMAP_FLAG_URGENT flag set.
UNSEEN	Match those messages which do not have the IMAP_FLAG_SEEN flag set.

In addition to the listed keywords, the keyword NOT may prefix a keyword to return those messages which do not match the search criteria. For example, "NOT TO user@domain.com" would return those messages which were not addressed to user@domain.com.

If multiple search keywords are specified, the result is the intersection of all those messages which meet the search criteria. For example, a search criteria of "DELETED SINCE 1-Jan-2003" would return all those messages which are marked for deletion and were added to the mailbox after 1 January 2003.

Those search keywords which expect dates must be specified in format ***dd-mmm-yyyy*** where the month is the three letter abbreviation for the month name. Note that the internal date the message was added to the mailbox is not the same as the value of the Date header field in the message.

The UID keyword expects a one or more unique message identifiers. These values may provided as comma separated list, or a range delimited by a colon. For example, "UID 23000:24000" would return all those messages who have UIDs ranging from 23000 through to 24000.

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDeletedMessages, GetMessageCount, GetMessageFlags, GetNewMessages, GetUnseenMessages

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::SelectMailbox Method

```
INT SelectMailbox(
   LPCTSTR lpszMailbox,
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **SelectMailbox** method selects the specified mailbox for read-write access.

Parameters

lpszMailbox

A null-terminated string which specifies the new mailbox to be selected.

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SelectMailbox** method is used to select a mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to selecting the new mailbox.

If you need to select a mailbox which contains Unicode characters or symbols, it is recommended you use the Unicode character set when compiling your application. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. If you do not use Unicode, this method will only accept mailbox names using modified UTF-7 encoding or UTF-8 encoding.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **SelectMailbox** again with the same mailbox name. Note that this will not cause any messages marked for deletion to be expunged.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-only mode, use the **ExamineMailbox** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMailbox, ExamineMailbox, GetFirstMailbox, GetMailboxStatus, GetNextMailbox, RenameMailbox,

ReselectMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::SetMessageFlags Method

```
INT SetMessageFlags(
   UINT nMessageId,
   UINT nMode
   DWORD dwMessageFlags
);
```

The **SetMessageFlags** method returns the message flags for the specified message.

Parameters

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

nMode

An unsigned integer value which specifies one of the following modes which determines how the message flags are set:

Value	Description
IMAP_FLAGS_REPLACE	All message flags are replaced with the flags specified by the ***dwMessageFlags*** parameter.
IMAP_FLAGS_ADD	The message flags specified by the ***dwMessageFlags*** parameter will be set for the message. Message flags that have been previously set will remain unmodified.
IMAP_FLAGS_REMOVE	The message flags specified by the ***dwMessageFlags*** parameter will be removed from the message. Message flags that are not specified will remain unmodified.

dwMessageFlags

An unsigned integer value which specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the method succeeds, it returns IMAP_RESULT_OK. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

DeleteMessage, GetMessageCount, GetMessageFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::StoreMessage Method

```
INT StoreMessage(
   UINT nMessageId,
   LPCTSTR lpszFileName
);
```

The **StoreMessage** method retrieves a message from the current mailbox and stores it in a local file or the system clipboard.

Parameters

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreMessage** method provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This method always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

If event handling is enabled, the IMAP_EVENT_PROGRESS event will fire periodically during the transfer of the message to the local system. An application can determine how much of the message has been retrieved by calling the **GetTransferStatus** method.

To retrieve the message into a global memory buffer so that it can be passed to the MIME or SMTP libraries, use the **GetMessage** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessage, GetMessageHeaders, GetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::SubscribeMailbox Method

```
INT SubscribeMailbox(
   LPCTSTR lpszMailbox
);
```

The **SubscribeMailbox** method subscribes the user to the specified mailbox.

Parameters

lpszMailbox

A null-terminated string which specifies the mailbox to subscribe to.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SubscribeMailbox** method adds the specified mailbox to the current user's list of active or subscribed mailboxes. The user will remain subscribed to the mailbox across multiple sessions, until the **UnsubscribeMailbox** method is called to remove the mailbox from the subscription list.

To list those mailboxes which the user has subscribed to, use the **GetFirstMailbox** method and specify the IMAP_LIST_SUBSCRIBED option.

Note that if a user subscribes to a mailbox and that mailbox is later renamed or deleted, the mailbox will not be automatically removed from the user's subscription list. An application must not assume that because a mailbox name is included in the list of subscribed mailboxes, it exists and can be selected. To check if the mailbox exists, use the **GetMailboxStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ExamineMailbox, GetFirstMailbox, GetNextMailbox, SelectMailbox, UnselectMailbox, UnsubscribeMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::UndeleteMessage Method

```
INT UndeleteMessage(
   UINT nMessageId
);
```

The **UndeleteMessage** method removes the deletion flag for the specified message.

Parameters

nMessage

Number of message to undelete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

DeleteMessage, GetDeletedMessages, GetMessage, GetMessageCount, GetMessageFlags, ReselectMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::UnselectMailbox Method

```
INT UnselectMailbox(
   BOOL bExpunge
);
```

The **UnselectMailbox** method unselects the current mailbox.

Parameters

bExpunge

A boolean flag which determines if deleted messages will be expunged from the mailbox. A non-zero value specifies that messages that have been marked for deletion will be removed from the mailbox. A zero value specifies that no messages will be removed from the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

DeleteMailbox, ExamineMailbox, GetFirstMailbox, GetMailboxStatus, GetNextMailbox, RenameMailbox, ReselectMailbox, SelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::UnsubscribeMailbox Method

```
INT UnsubscribeMailbox(
   LPCTSTR lpszMailbox
);
```

The **UnsubscribeMailbox** method unsubscribes the user from the specified mailbox.

Parameters

lpszMailbox

A null-terminated string which specifies the mailbox to unsubscribe from.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **UnsubscribeMailbox** method removes the specified mailbox from the current user's list of active or subscribed mailboxes.

To list those mailboxes which the user has subscribed to, use the **GetFirstMailbox** method and specify the IMAP_LIST_SUBSCRIBED option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ExamineMailbox, GetFirstMailbox, GetNextMailbox, SelectMailbox, SubscribeMailbox, UnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CImapClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string is passed as the buffer argument. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol Data Structures

- IMAPMAILBOX
- IMAPMAILBOXSTATUS
- IMAPMESSAGE
- IMAPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPMAILBOX

This structure contains information about a selected mailbox.

```
typedef struct _IMAPMAILBOX
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessageId;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwFlags;
    DWORD   dwPermanentFlags;
    DWORD   dwAccessMode;
    DWORD   dwReserved1;
    DWORD   dwReserved2;
} IMAPMAILBOX, *LPIMAPMAILBOX;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessageId

A value which specifies the message ID of the first unseen message in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwFlags

A value which specifies one or more mailbox flags. One or more of the following values may be specified:

Value	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any child mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this

	flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

dwPermanentFlags

A value which specifies the message flags that a client can change permanently. If this value is zero, then no permanent flags are defined for the mailbox and the client may assume that all message flags may be set permanently. Otherwise, one or more of the following values may be specified:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

dwAccessMode

A value which specifies the access mode for the mailbox. It may be one of the following values:

Value	Description
IMAP_ACCESS_READONLY	The mailbox has been selected in read-only mode. Messages may not be created in the mailbox, nor can message flags be modified.
IMAP_ACCESS_READWRITE	The mailbox has been selected in read-write mode. Messages may be modified by the client, and messages marked for deletion can be expunged.

dwReserved1

A reserved value that is undefined.

dwReserved2

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPMAILBOXSTATUS

This structure contains information about a mailbox.

```
typedef struct _IMAPMAILBOXSTATUS
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessages;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwReserved;
} IMAPMAILBOXSTATUS, *LPIMAPMAILBOXSTATUS;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessages

A value which specifies the number of unread messages in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPMESSAGE

This structure contains information about a message.

```
typedef struct _IMAPMESSAGE
{
    UINT    nMessageId;
    DWORD   dwMessageUID;
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwTimestamp;
    DWORD   dwReserved;
} IMAPMESSAGE, *LPIMAPMESSAGE;
```

Members

nMessageId

An sequential integer value which identifies the message. The message identifier is only valid while the mailbox is selected and no messages marked for deletion have been expunged. To maintain a persistent identifier for the message, use a combination of the mailbox UID and message UID.

dwMessageUID

An integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque. If the client application stores the message UID on the local system, it should also store the UID for the mailbox that contains the message. If the mailbox UID changes, the message UID may no longer be valid.

dwSize

Specifies the size of the message in bytes. This value may not be the same as the number of characters in the message, particularly if the message text uses Unicode or extended multibyte ANSI sequences. For a structured multipart MIME message, this includes the combined size of all message parts.

dwFlags

A value which specifies one or more message flags. One or more of the following values may be specified:

Value	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

dwTimestamp

An integer value which specifies the date and time that the message was added to the mailbox. The value is expressed as the number of seconds since midnight, 1 January 1970 and is the same value that is used for the standard C runtime library time methods. Note that the date and time used is

the message's internal date from the mail server, not the value of the **Date** header field in the message.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPTRANSFERSTATUS

This structure is used by the GetTransferStatus method to return information about a file transfer in progress.

```
typedef struct _IMAPTRANSFERSTATUS
{
   UINT    nMessageId;
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} IMAPTRANSFERSTATUS, *LPIMAPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Server Class Library

A general purpose TCP/IP networking library for developing server applications.

Reference

- Data Members
- Class Methods
- Event Handlers
- Data Structures
- Error Codes

Library Information

Class Name	CInternetServer
File Name	CSWSKV12.DLL
Version	12.0.3024.1286
LibID	99EEE601-E935-4266-B046-771A823DB47D
Import Library	CSWSKV12.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

The Internet Server class library provides a simplified interface for creating event-driven, multithreaded server applications using the TCP/IP protocol. Each instance of the Internet Server class represents a server, and each active client connection is managed internally and referenced by a handle which uniquely identifies the client session. The class library supports secure connections using the standard TLS protocol and can be used to create secure, custom server programs.

This class is designed to be used as a base class from which your own server class is derived. To exchange data with the clients that connect to the server, you should override the default events such as **OnConnect** and **OnRead**. Most interaction with the clients occur within these event handlers. Because the client sessions are managed in worker threads that are separate from the main UI thread of your application, you may perform a blocking operation in response to an event without affecting the other clients that are connected to the server.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must

have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer Public Data Members

Member Variables	Description
m_nBacklog	The size of the backlog connection queue for the server
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_dwOptions	The options specified when creating an instance of the server
m_nPriority	The priority specified when creating an instance of the server
m_dwStackSize	The initial size of the stack allocated for threads created by the server
m_nTimeout	The timeout period in seconds waiting for a blocking operation to complete

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_nBacklog

```
UINT m_nBacklog;
```

The size of the backlog connection queue for the server.

Remarks

The **m_nBacklog** data member is a public variable that specifies the size of the queue allocated for pending client connections. A value of zero specifies that the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CInternetServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_nMaxClients

```
UINT m_nMaxClients;
```

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

CInternetServer, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_nMaxClientsPerAddress

```
UINT m_nMaxClientsPerAddress;
```

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

See Also

CInternetServer, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_dwOptions

```
DWORD m_dwOptions;
```

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see Server Option Constants. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CInternetServer, GetOptions, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_nPriority

```
INT m_nPriority;
```

The priority specified when creating an instance of the server.

Remarks

The **m_nPriority** data member is a public variable that specifies the which specifies the priority for the server and all client sessions. Changing the value of this data member does not have an effect on an active instance of the server. It may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It can be used when it is important for the client session thread to be highly responsive. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The thread will be given higher scheduling priority and will be more responsive to the remote host. It is not recommended that this priority be used on a system with a single processor.

See Also

CInternetServer, GetPriority, SetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_dwStackSize

```
DWORD m_dwStackSize;
```

The initial size of the stack allocated for threads created by the server.

Remarks

The **m_dwStackSize** data member is a public variable that specifies the initial amount of memory that is committed to the stack for each thread created by the server. A value of zero specifies that the default stack size should be used, which is 256K for 32-bit processes and 512K for 64-bit processes. This variable can be modified directly or by calling the **SetStackSize** method. Changing the value of this data member does not have an effect on an active instance of the server. It is recommended that most applications use the default stack size.

See Also

CInternetServer, GetStackSize, SetStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::m_nTimeout

```
DWORD m_nTimeout;
```

The default options used when starting an instance of the server.

Remarks

The **m_nTimeout** data member is a public variable that specifies the number of seconds the server should wait for a client to perform a network operation. If the client does not exchange any information with the server within this period of time, a timeout event will occur. The timeout value affects all clients that are connected to the server. This variable can be modified directly or by calling the **SetTimeout** method. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

CInternetServer, GetTimeout, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer Class Methods

Class	Description
CInternetServer	Constructor which initializes the current instance of the class
~CInternetServer	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the connection and immediately close the socket
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
Broadcast	Write data to all active clients currently connected to the server
Cancel	Cancel a blocking operation for the specified client session
CompareAddress	Compare two IP addresses to determine if they are identical
DetachHandle	Detach the server handle from the current instance of this class
DisableSecurity	Disable secure communication with the client
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect the client, closing the socket handle and terminating the session
EnableSecurity	Enable secure communication with the client
EnableTrace	Enable logging of network function calls to a file
EnumClients	Returns a list of active client connections established with the server
EnumNetworkAddresses	Return the list of network addresses that are configured for the local host
FindClient	Returns a handle to the client which matches the specified client ID or moniker
FormatAddress	Convert an IP address in binary format into a printable string
Flush	Flush the send and receive buffers for the specified client session
GetActiveClient	Return the socket handle for the active client session
GetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
GetAddress	Convert an IP address string to a binary format
GetAddressFamily	Return the address family for the specified IP address
GetBacklog	Return the size of the backlog connection queue for the server
GetClientAddress	Return the IP address and port number for the specified client session
GetClientData	Returns the application defined data associated with the specified client session
GetClientHandle	Returns the handle for a specific client session based on its ID number
GetClientId	Returns the unique ID number assigned to the specified client session
GetClientIdleTime	Returns the amount of time the specified client session has been idle
GetClientMoniker	Returns the string alias associated with the specified client session
GetClientPort	Returns the remote port number used by the client to establish the connection
GetClientServer	Returns a socket handle to the server for the specified client socket
GetClientServerById	Returns a socket handle to the server for the specified session identifier

GetClientThreadId	Returns the thread ID for the specified client
GetClientThreads	Returns the number of client session threads created by the server
GetErrorString	Return a description for the specified error code
GetExternalAddress	Return the external IP address assigned to the local system
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the IP address assigned to the specified hostname
GetHostName	Return the hostname assigned to the specified IP address
GetLastError	Return the last error code
GetLocalAddress	Return the local IP address and port number for the server
GetLocalName	Return the hostname assigned to the local system
GetMemoryUsage	Return the amount of memory allocated for the server
GetOptions	Return the current server options
GetPriority	Return the current priority assigned to the server
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetStatus	Return the current status of the server
GetStreamInfo	Return information about the current stream I/O operation
GetThreadClient	Return the handle for the client session that is being managed by the specified thread
GetTimeout	Return the timeout interval for blocking operations in seconds
IsActive	Determine if the server is currently active
IsAddressNull	Determine if the specified IP address is a null address
IsAddressRoutable	Determine if the specified IP address is routable over the Internet
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the server is listening for client connections
IsLocked	Determine if the server is currently in a locked state
IsProtocolAvailable	Determine if the specified protocol and address family are supported
IsReadable	Determine if data is available to be read from the client
IsWritable	Determine if data can be sent to the client without causing the thread to block
Lock	Lock the server, causing all other client threads to block until it is unlocked
MatchHostName	Match a host name against one more strings that may contain wildcards
Peek	Read data from the client without removing it from the socket buffer
Read	Read data from the client
ReadLine	Read a line of data from the client, storing it in a string buffer
ReadStream	Read a stream of data from the client and store it in the specified buffer
Reject	Reject a pending client connection
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SetBacklog	Set the size of the backlog connection queue for the server

SetCertificate	Specify the server certificate that should be used with secure connections
SetClientData	Associate application defined data with the specified client session
SetClientMoniker	Associate a unique string alias with the specified client session
SetLastError	Set the last error code
SetOptions	Set one or more server options
SetPriority	Set the priority assigned to the server
SetTimeout	Set the timeout interval used when waiting for a blocking operation to complete
ShowError	Display a message box with a description of the specified error
Start	Begin listening for client connections on the specified address and port
Stop	Stop listening for connections and terminate all client sessions
StoreStream	Read a stream of data from the client and store it in a file
Suspend	Suspend accepting client connections and optionally reject or disconnect clients
Throttle	Limit the number of active client connections, connections per address and connection rate
Unlock	Unlock the server, allowing other client threads to resume execution
ValidateCertificate	Validate the specified security certificate is installed on the local system
Write	Write data to the client
WriteLine	Write a line of data to the client, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the client

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::CInternetServer Method

```
CInternetServer();
```

The **CInternetServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **InetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

~CInternetServer, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::~CInternetServer

```
~CInternetServer();
```

The **CInternetServer** destructor releases resources allocated by the current instance of the **CInternetServer** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CInternetServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the connection are destroyed. If there are any clients connected to the server at the time the destructor is called, those client sessions will be immediately terminated.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CInternetServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Abort Method

```
BOOL Abort(
   SOCKET hSocket
);

BOOL Abort();
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Abort** method should only be used when the connection must be closed immediately. This method should only be used to abort client connections and should not be called within an **OnAccept** event handler. To reject an incoming client connection, use the **Reject** method.

In most cases, the server should call the **Disconnect** method to gracefully close a client connection. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Cancel, Disconnect, Reject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::AsyncNotify Method

```
BOOL AsyncNotify(
   HWND hWnd,
   UINT uMsg
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the ***hWnd*** parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the ***hWnd*** parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The ***wParam*** message parameter will contain the notification message and the ***lParam*** message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Value	Description
INET_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the **Start** method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The ***lParam*** message parameter will specify the handle to the server.
INET_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the **Resume** method to resume

	accepting client connections. The ***lParam*** message parameter will specify the handle to the server.
INET_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the **Suspend** method. This notification message may be sent to the application multiple times over the lifetime of the server. The ***lParam*** message parameter will specify the handle to the server.
INET_NOTIFY_RESTART	This notification is sent when the server is restarted using the **Restart** method. Note that the server socket handle provided by the ***lParam*** message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The ***lParam*** message parameter will specify the handle to the server.
INET_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The ***lParam*** message parameter will specify the unique ID of the client that connected to the server.
INET_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the **Stop** method. The ***lParam*** message parameter will specify the unique ID of the client that disconnected from the server.
INET_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in ***lParam*** will reference the defunct server and cannot be used with other server methods. The ***lParam*** message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The INET_NOTIFY_CONNECT and INET_NOTIFY_DISCONNECT notifications are different from the other server notifications because the ***lParam*** message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. If you need to obtain the handle to the client session using the ID, call the **GetClientHandle** method. To obtain the server handle in response to the INET_NOTIFY_CONNECT message, use the **GetClientServerById** method. Note that at the time the application processes the INET_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientHandle, GetClientServerById

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::AttachHandle Method

```
VOID AttachHandle(
   SOCKET hSocket
);
```

Attach the specified server socket handle to the current instance of the class.

Parameters

hSocket

The socket handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketWrench API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle has already been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the ***hSocket*** parameter is presumed to be a valid server socket handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid socket handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Broadcast Method

```
INT Broadcast(
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

Sends data to all clients that are connected to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the method succeeds, the return value is the number of clients that the data was sent to. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **Broadcast** method sends the contents of the buffer to all of the clients that are connected to the server. This method will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Cancel Method

```
BOOL Cancel(
   SOCKET hSocket
);
```

Cancel a blocking operation for the specified client session.

Parameters

hSocket

The handle to a client socket.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation using the same client socket handle. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Canceling a blocking operation for another client session may yield unpredictable results. If you wish to terminate the client session, it is preferable to use the **Disconnect** method rather than using this method in conjunction with the **Abort** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Abort, Disconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::CompareAddress Method

```
BOOL CompareAddress(
   LPINTERNET_ADDRESS lpAddress1,
   LPINTERNET_ADDRESS lpAddress2
);

BOOL CompareAddress(
   LPCTSTR lpszAddress1,
   LPCTSTR lpszAddress2
);
```

Compare two IP addresses to determine if they are identical.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

Return Value

If the method succeeds and the two addresses are identical, the return value is non-zero. If the method fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::DetachHandle Method

```
SOCKET DetachHandle();
```

The **DetachHandle** method detaches the server socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server socket associated with the current instance of the class object. If the server is not active, the value INVALID_SOCKET will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketWrench API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::DisableSecurity Method

```
BOOL DisableSecurity(
   SOCKET hSocket
);

BOOL DisableSecurity();
```

Disable secure communication with the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DisableSecurity** method disables a secure session, with subsequent calls to **Read** and **Write** sending and receiving unencrypted data. It is important to note that because this method sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This method does not close the socket. Use the **Disconnect** method to close the socket and release the resources allocated for the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnableSecurity, SetCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Disconnect Method

```
BOOL Disconnect(
   SOCKET hSocket
);

BOOL Disconnect();
```

Disconnect the client, closing the socket handle and terminating the session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

Once the connection has been terminated, the client socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

The **Disconnect** method should only be used to terminate client sessions and the server handle should never be provided as the ***hSocket*** parameter. To stop the server, use the **Stop** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Abort, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::EnableSecurity Method

```
BOOL EnableSecurity(
   SOCKET hSocket
);

BOOL EnableSecurity();
```

Enable secure communication with the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableSecurity** method enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate specified by a previous call to the **SetCertificate** method. This method will cause the calling thread to block and wait for the client to initiate the TLS handshake.

This method is typically used to implement support for explicit TLS connections, where the client establishes a standard, non-secure connection to the server and then negotiates a secure connection at a later point. Usually this is done by the client sending a specific command to the server, and the server calls **EnableSecurity** from within the **OnRead** event handler that processes the command. If the method succeeds, all subsequent calls to **Read** and **Write** to receive and send data will be encrypted.

This method is only used to enable a secure connection for a specific client session. If all client connections should be secure, then call the **SetOptions** method to specify the INET_OPTION_SECURE option prior to starting the server and call the **SetCertificate** method to specify the server certificate that should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

DisableSecurity, SetCertificate, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the process level and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "`0x`". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::EnumClients Method

```
INT EnumClients(
   INTERNET_ADDRESS& ipAddress,
   SOCKET *lpClients,
   INT nMaxClients
);

INT EnumClients(
   LPCTSTR lpszAddress,
   SOCKET *lpClients,
   INT nMaxClients
);

INT EnumClients(
   SOCKET *lpClients,
   INT nMaxClients
);
```

Return a list of active client connections established with the server.

Parameters

ipAddress

A reference to an INTERNET_ADDRESS structure that contains the IP address that should be matched against the clients connected to the server. Only those clients that have connected to the server from this address will be returned in the ***lpClients*** array.

lpszAddress

A string that specifies the IP address that should be matched against the clients connected to the server. Only those clients that have connected to the server from this address will be returned in the ***lpClients*** array.

lpClients

Pointer to an array which will contain the socket handle for each active client session when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of socket handles to be returned in the ***lpClients*** array. If the ***lpClients*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. A return value of zero indicates that there are either no active client sessions, or no clients have connected using the specified IP address. If the method fails, the return value is FTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections, the method will fail. To dynamically determine the number of active connections, call the method with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero.

This method will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this method will return zero,

indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server. To determine the actual number of client sessions regardless of their status, use the **GetClientThreads** method.

Example

```
// Populate a listbox with the IP address for each client
pListBox->ResetContent();

INT nClients = pServer->EnumClients();
if (nClients > 0)
{
    SOCKET *phClients = new SOCKET[nClients];

    nClients = pServer->EnumClients(phClients, nClients);
    if (nClients == INET_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        CString strAddress;

        if (pServer->GetClientAddress(phClients[nIndex], strAddress))
            pListBox->AddString(strAddress);
    }

    // Free the memory allocated for the socket handles
    delete phClients;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientAddress, GetClientThreads

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::EnumNetworkAddresses Method

```
INT EnumNetworkAddresses(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddressList,
   INT nMaxAddresses
);

INT EnumNetworkAddresses(
   LPINTERNET_ADDRESS lpAddressList,
   INT nMaxAddresses
);
```

The **EnumNetworkAddresses** method returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses for the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that support IPv6.
INET_ADDRESS_IPV4	Specifies that the addresses should be in IPv4 format. The first four bytes of the *ipNumber* array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the addresses should be in IPv6 format. All bytes in the *ipNumber* array are significant.

lpAddressList

A pointer to an array of INTERNET_ADDRESS structures that will contain the IP address of each local network interface.

nMaxAddresses

Maximum number of addresses to be returned.

Return Value

If the function succeeds, the return value is the number of network addresses that are configured for the local host. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

If this method is called without specifying the address family, the value INET_ADDRESS_IPV4 is used. This is provided primarily for compatibility with legacy applications and it is recommended that you explicitly specify the address family.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

FormatAddress, GetHostAddress, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::FindClient Method

```
SOCKET FindClient(
   UINT nClientId
);

SOCKET FindClient(
   LPCTSTR lpszMoniker
);
```

Returns a handle to the client which matches the specified client ID or moniker.

Parameters

nClientId

An unsigned integer the specifies a unique client ID for the session. This value must be greater than zero.

lpszMoniker

A null-terminated string which specifies the client moniker to search for. This parameter cannot be NULL and cannot specify an empty string.

Return Value

If the method succeeds, the return value is the handle to the client socket for the session that matches the specified client ID or moniker. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **SetClientMoniker** method. This method will search all active client sessions for the server, and returns the socket handle to the client that matches the specified moniker. If there is no match, an error will be returned.

The moniker can be any string value, however monikers are not case sensitive and may not contain embedded null characters. The maximum length of a moniker is 127 characters.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientMoniker, SetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Flush Method

```
BOOL Flush(
   SOCKET hSocket
);

BOOL Flush();
```

Flush the send and receive buffers for the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** method or prior to calling the **Disconnect** method.

An application never needs to use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::FormatAddress Method

```
INT FormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszAddress,
   INT cchAddress
);

INT FormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   CString& strAddress
);
```

The **FormatAddress** method converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

The buffer that will contain the formatted IP address. This buffer should be at least 46 characters in length. This may also reference a **CString** object which will contain the formatted address when the method returns.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the IP address string. If the method fails, the return value is INET_ERROR, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the INTERNET_ADDRESS structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetLocalAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetActiveClient Method

```
SOCKET GetActiveClient();
```

Return the socket handle for the active client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the socket handle for the active client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetActiveClient** method returns a handle to the client socket for the active client session. The active session is determined by the session thread that is currently executing, and therefore is only meaningful within the context of a server event handler such as **OnConnect** or **OnRead**. The value returned by the his method is the same as the client socket handle that is passed to the event handler.

This method will fail within an **OnAccept** event handler because at that point the connection has not yet been accepted, therefore there is no active client session. It will also fail if called outside of an event handler. To obtain the socket handle associated with a particular session thread, use the **GetThreadClient** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientHandle, GetClientThreadId, GetThreadClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetAdapterAddress Method

```
INT GetAdapterAddress(
   INT nAdapterIndex,
   INT nAddressType,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetAdapterAddress(
   INT nAdapterIndex,
   INT nAddressType,
   CString& strAddress
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Value	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type. An alternate form of the method accepts a **CString** argument which will contain the address.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the method fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **GetLastError**.

Remarks

The **GetAdapterAddress** method will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this method would return zero.

This method will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this method may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    CString strAddress;

    if (pServer->GetAdapterAddress(nIndex, INET_ADAPTER_IPV4, strAddress) ==
INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnumNetworkAddresses, GetLocalAddress, GetLocalName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetAddress Method

```
INT GetAddress(
   LPCTSTR lpszAddress,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);

INT GetAddress(
   LPCTSTR lpszAddress,
   LPINTERNET_ADDRESS lpAddress
);
```

The **GetAddress** method converts an IP address string to binary format.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the ***lpszAddress*** parameter. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the ***lpszAddress*** parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the ***lpszAddress*** parameter does not specify a valid IPv4 address string, this method will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the ***ipNumber*** array are significant. If the ***lpszAddress*** parameter does not specify a valid IPv6 address string, this method will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member

should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the ***lpszAddress*** parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, IsAddressNull, IsAddressRoutable, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetAddressFamily Method

```
INT GetAddressFamily(
   LPCTSTR lpszAddress,
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the method succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the method fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call the **GetLastError** method.

Value	Description
INET_ADDRESS_IPV4	The address passed to the method is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the method is a valid IPv6 address.

Remarks

The **GetAddressFamily** method returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other methods such as **Connect**. Note that this method will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsAddressNull, IsAddressRoutable, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetBacklog Method

```
UINT GetBacklog();
```

Return the size of the backlog connection queue for the server.

Parameters

None.

Return Value

The return value is the size of the queue used to accept client connections.

Remarks

The **GetBacklog** method returns the size of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. To change the size of the backlog queue, use the **SetBacklog** method prior to starting the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

SetBacklog, Start, Data Members

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientAddress Method

```
INT GetClientAddress(
   SOCKET hSocket,
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnRemotePort
);

INT GetClientAddress(
   SOCKET hSocket,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetClientAddress(
   SOCKET hSocket,
   CString& strAddress
);
```

Return the IP address and port number for the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to a server or client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the client that is connected to the server. This parameter may be NULL, in which case the IP address will not be returned to the caller.

lpnRemotePort

A pointer to an unsigned integer that will contain the port number of the client that is connected to the server. This parameter may be NULL, in which case the port number will not be returned to the caller.

lpszAddress

A null-terminated string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

nMaxLength

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If this method is called within an **OnAccept** event handler, passing the server handle as the ***hSocket*** parameter will return the IP address and port number for the client that is attempting to establish the connection. If the client address is unavailable, the ***ipFamily*** member of the INTERNET_ADDRESS structure will be zero.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientPort, OnAccept, OnConnect, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientData Method

```
BOOL GetClientData(
   SOCKET hClient,
   LPVOID *lppvData
);

BOOL GetClientData(
   LPVOID *lppvData
);
```

Returns the application defined data associated with the specified client session.

Parameters

hClient

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lppvData

A pointer to a void pointer which will contain an application defined value associated with the client session.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be retrieved. To get extended error information, call **GetLastError**.

Remarks

The **GetClientData** method is used to retrieve the application defined data that was previously associated with a client session using the **SetClientData** method. This is typically used to associate a pointer to a data structure or a class instance with a specific client handle.

This method can only be used with client socket handles created using the server interface. If the socket handle is invalid, or does not reference a client socket handle created by the server, the *lppvData* pointer passed to this method will be initialized to a value of NULL and the method will return a value of zero.

If this method is called with a valid socket handle and there is no data associated with the socket, the method will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this method, the application should always check the return value to ensure the method succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = new UINT;
UINT *pnValue2 = NULL;

*pnValue1 = 1234;

if (!pServer->SetClientData(hClient, pnValue1))
{
    // Unable to associate the data with this session
    return;
}
```

```
if (!pServer->GetClientData(hClient, (LPVOID *)&pnValue2))
{
    // Unable to retrieve the data associated with this session
    return;
}

// pnValue2 == pnValue1
printf("The value of the user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

SetClientData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientHandle Method

```
SOCKET GetClientHandle(
   UINT nClientId
);
```

Returns the handle for a specific client session based on its ID number.

Parameters

nClientId

An unsigned integer value which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the socket handle for the specified client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value called the client ID. The **GetClientHandle** method will return the socket handle that is associated with the specified client ID. Unlike socket handles, which are reused by the operating system, the client ID is guaranteed to be unique throughout the lifetime of the server. To obtain the ID associated with the client session, use the **GetClientId** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientId, GetClientMoniker, SetClientMoniker, GetThreadClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientId Method

```
UINT GetClientId(
   SOCKET hSocket
);

UINT GetClientId();
```

Returns the unique ID number assigned to the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is an unsigned integer value which uniquely identifies the client session. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **GetClientId** method and used by the application to identify that client session. The **GetClientHandle** method can then be used to used to obtain the client socket handle for the session, based on that client ID. It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientHandle, GetClientMoniker, SetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientIdleTime Method

```
DWORD GetClientIdleTime(
   SOCKET hSocket
);

DWORD GetClientIdleTime();
```

Returns the number of milliseconds that the specified client session has been idle.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is an unsigned integer value which specifies the number of milliseconds the client session has been idle. If the method fails, the return value is INFINITE. To get extended error information, call **GetLastError**.

Remarks

The **GetClientIdleTime** method will return the number of milliseconds that have elapsed since data was exchanged with the client. The elapsed time is limited to the resolution of the system timer, which is typically in the range of 10 milliseconds to 16 milliseconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientHandle, GetClientId, GetClientMoniker, GetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientMoniker Method

```
INT GetClientMoniker(
   SOCKET hClient,
   LPTSTR lpszMoniker,
   INT nMaxLength
);

INT GetClientMoniker(
   SOCKET hClient,
   CString& strMoniker
);
```

The **GetClientMoniker** method returns the moniker associated with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszMoniker

Pointer to a string buffer that will contain the moniker for the specified client session when the method returns. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of characters that may be copied into the string buffer. The buffer must be large enough to store the moniker and a terminating null character. The maximum length of a moniker is 127 characters.

Return Value

If the method succeeds, the return value is the number of characters in the moniker string. A return value of zero specifies that no moniker was assigned to the socket. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **SetClientMoniker** method. This method will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **FindClient** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

FindClient, GetClientId, SetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientPort Method

```
INT GetClientPort(
   SOCKET hSocket
);

INT GetClientPort();
```

Returns the remote port number used by the client to establish the connection.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is the port number. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetClientPort** method returns the remote port number that the client is bound to. Note that this is not the port number that the server is using to listen for connections, it is the port number that the client is bound to on the remote host. Typically this is an ephemeral port, either in the range of 1025 through 5000, or greater than 32768, depending on the client operating system.

If this method is called within the **OnAccept** event handler, providing the server socket handle as the *hSocket* parameter will return the port number of the client that is attempting to establish the connection.

It is not recommended that you use the client port number for anything other than informational and logging purposes. Do not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientAddress, OnAccept, OnConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientServer Method

```
SOCKET HttpGetClientServer(
    UINT nClientId
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return INVALID_SOCKET indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AsyncNotify

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientServerById Method

```
SOCKET GetClientServerById(
    UINT nClientId
);
```

The **GetClientServerById** method returns a socket handle to the server for the specified client session identifier.

Parameters

nClientId

Client session identifier.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServerById** method returns the handle to the server that created the client session using the client's unique identifier. The **GetClientServer** method can be used to obtain the server handle using the client socket handle rather than the client session ID. This method is typically used in conjunction with the INET_NOTIFY_CONNECT notification message to obtain the handle to the server that generated the event using the client ID passed in the ***wParam*** message parameter.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AsyncNotify, GetClientHandle, GetClientId, GetClientServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientThreadId Method

```
DWORD GetClientThreadId(
   SOCKET hSocket
);

DWORD GetClientThreadId();
```

Returns the thread ID associated with the specified client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **Disconnect** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnumClients, GetActiveClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetClientThreads Method

```
INT GetClientThreads();
```

Returns the number of client session threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of client session threads that have been created by the server. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **InetGetClientThreads** function returns the number of threads that are managing client sessions for the specified server. If there are no clients connected to the server, this function will return a value of zero. Because this function returns the number of session threads, the value returned will include those clients that are in the process of disconnecting from the server but their session thread has not yet terminated. This differs from the **InetEnumServerClients** function which will only enumerate active clients.

If you wish to determine when the last client has disconnected from the server, call this function within an event handler for the INET_EVENT_DISCONNECT event. If the function returns a value greater than one, then there are other client sessions that are either connected or in the process of terminating.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnumClients

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetExternalAddress Method

```
INT GetExternalAddress(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress,
   INT nMaxLength
);

INT GetExternalAddress(
   INT nAddressFamily,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetExternalAddress(
   INT nAddressFamily,
   CString& strAddress
);
```

The **GetExternalAddress** method returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The method will attempt to determine the external IP address using an IPv4 network connection.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. The method will attempt to determine the external IP address using an IPv6 network connection and requires that the local host have an IPv6 network interface installed and enabled.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the external IP address of the local host in binary form.

lpszAddress

A null-terminated string buffer that will contain the external IP address of the local host.

nMaxLength

The maximum length of the string that will contain the IP address when the method returns.

Return Value

In the first form of the method, if it succeeds, the return value is the IP address of the local system in numeric form. If the method fails, the return value is INET_ADDRESS_NONE. In the second form, the return value is the length of the IP address string and an error is indicated by the return value INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetExternalAddress** method returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **GetLocalAddress** method will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **GetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

This method requires that you have an active connection to the Internet and calling this function on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. An application should always check the return value in case there is an error; never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this function only be used with broadband connections where a NAT router is being used.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a new thread and call this function from within that thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetHostAddress, GetLocalAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetHandle Method

```
SOCKET GetHandle();
```

The **GetHandle** method returns the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the socket handle associated with the current instance of the class object. If there is no active connection, the value INVALID_SOCKET will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketWrench API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetHostAddress Method

```
INT GetHostAddress(
   LPCTSTR lpszHostName,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);

INT GetHostAddress(
   LPCTSTR lpszHostName,
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetGetHostAddress** method resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

The name of the host to resolve; this may be a fully-qualified domain name or an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the *ipNumber* array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the *ipNumber* array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the specified host.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This method can also be used to convert an address in dot notation to a binary format. If the method must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future

compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the ***ipFamily*** member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The ***nAddressFamily*** parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the ***lpszHostName*** parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetHostName, GetLocalAddress, GetLocalName, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetHostName Method

```
INT GetHostName(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszHostName,
   INT cchHostName
);

INT GetHostName(
   LPINTERNET_ADDRESS lpAddress,
   CString& strHostName
);
```

The **GetHostName** method performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the IP address that should be resolved into a host name.

lpszHostName

The buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the hostname.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the method must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This method requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this method frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetHostAddress, GetLocalAddress, GetLocalName, IsProtocolAvailable,

INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetLocalAddress Method

```
INT GetLocalAddress(
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnPort
);

INT GetLocalAddress(
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetLocalAddress(
   CString& strAddress
   UINT *lpnPort
);
```

Return the local IP address and port number for the server.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the local host. If the server is not active, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpszAddress

A null-terminated string that will contain the IP address of the local host. If this version of the method is used, the IP address is converted to a string format using the **FormatAddress** method. The string should be able to store at least 46 characters to ensure that both IPv4 and IPv6 formatted addresses can be returned without the possibility of a buffer overrun. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If the server is active, this parameter will be set to the local port that the listening socket was bound to. If the server is not active, this parameter is ignored. If the port number is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **GetLocalAddress** method will return the local, non-routable IP address assigned to the local system. To determine the public IP address has been assigned to the system, you should use the **GetExternalAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetExternalAddress, GetHostAddress, GetHostName, GetLocalName, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetLocalName Method

```
INT GetLocalName(
   LPTSTR lpszHostName,
   INT cchHostName
);

INT GetLocalName(
   CString& strHostName
);
```

The **GetLocalName** method returns the hostname assigned to the local system.

Parameters

lpszHostName

The buffer that will contain the hostname. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the local hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientAddress, GetLocalAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of bytes allocated. If the method fails, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated for the server. It enumerates all heap allocations made by the process and returns the total number of bytes allocated for the server and all client sessions. This value only reflects the amount of memory explicitly allocated by this class library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the server into a locked state, and all network operations will block until the method returns. Because this method enumerates all heaps created for the process, it can be an expensive operation, particularly when a large number of connections have been established. Frequent use of this method can significantly degrade the performance of the application. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetOptions Method

```
DWORD GetOptions();
```

Return the current server options.

Parameters

None.

Return Value

This method returns an unsigned integer value that specifies the server options that are currently enabled for the class instance. For a list of the available options, see Server Option Constants. This method returns the value of the **m_dwOptions** class member variable.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

SetOptions, Data Members

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetPriority Method

```
INT GetPriority();
```

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

SetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetStackSize Method

```
DWORD GetStackSize();
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

SetStackSize, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetStatus Method

```
INT GetStatus();
```

Return the current status of the server.

Parameters

None.

Return Value

An integer value that specifies the server status.

Remarks

The return value is one of the following values:

Value	Description
INET_SERVER_INACTIVE	The server is currently inactive.
INET_SERVER_STARTED	The server has initialized and is preparing to listen for client connections.
INET_SERVER_LISTENING	The server is actively listening for incoming client connections.
INET_SERVER_SUSPENDED	The server has been suspended and is no longer accepting client connections.
INET_SERVER_SHUTDOWN	The server has been stopped and is in the process of terminating all active client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsInitialized, IsListening, IsLocked

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetStreamInfo Method

```
BOOL GetStreamInfo(
   SOCKET hSocket
   LPINETSTREAMINFO lpStreamInfo
);

BOOL GetStreamInfo(
   LPINETSTREAMINFO lpStreamInfo
);
```

The **GetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpSecurityInfo

A pointer to an **INETSTREAMINFO** structure which contains information about the status of the current operation.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStreamInfo** method returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this method will return the status of the last successful streaming read or write performed by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

ReadStream, StoreStream, WriteStream, INETSTREAMINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetThreadClient Method

```
SOCKET WINAPI InetGetThreadClient(
    DWORD dwThreadId
);
```

The **GetThreadClient** method returns the socket handle for the client session that is being managed by the specified thread.

Parameters

dwThreadId

An unsigned integer value which identifies the thread managing the client session. If this parameter has a value of zero, then the client handle for the current thread is returned.

Return Value

If the method succeeds, the return value is the socket handle for the specified client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **GetThreadClient** is used to obtain the socket handle for the client session that is being managed by the specified thread. If the specified thread ID is zero, then the method will return the client socket for the current thread, otherwise it will search the internal table of all active client sessions and return the handle to the session that is being managed by that thread.

This method will fail if the thread ID does not specify an active client session thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetActiveClient GetClientHandle, GetClientId, GetClientThreadId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::GetTimeout Method

```
INT GetTimeout();
```

Return the timeout interval for blocking network operations in seconds.

Parameters

None.

Return Value

The return value is the timeout period in seconds. If there is no active server, this will return the timeout period that will be used when the server is started. A value of zero specifies that a reasonable default timeout period will be automatically selected.

Remarks

The **GetTimeout** method returns the number of seconds the server will wait for a blocking network operation to complete, such as sending or receiving data. This value also determines the amount of time that the server will wait for the client to send data before invoking the **OnTimeout** event handler for that session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, SetTimeout, OnTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsActive Method

```
BOOL IsActive();
```

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CInternetServer, IsListening, Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsAddressNull Method

```
BOOL IsAddressNull(
   LPCTSTR lpszAddress
);

BOOL IsAddressNull(
   LPINTERNET_ADDRESS lpAddress
);
```

The **IsAddressNull** method determines if the IP address is null.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is null, or the parameter is a NULL pointer, the return value is non-zero. If the method fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAddress, IsAddressRoutable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsAddressRoutable Method

```
BOOL IsAddressRoutable(
   LPCTSTR lpszAddress
);
```

```
BOOL IsAddressRoutable(
   LPINTERNET_ADDRESS lpAddress
);
```

The **IsAddressRoutable** method determines if the IP address is routable over the Internet.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is routable over the Internet, the return value is non-zero. If the method fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAddress, GetExternalAddress, IsAddressNull, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CInternetServer, IsListening, IsLocked

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsListening Method

```
BOOL IsListening();
```

Determine if the server is listening for client connections.

Parameters

None.

Return Value

If the server has started and is listening for client connections, the method returns a non-zero value. If the server is not listening for connections, the return value is zero.

Remarks

The **IsListening** method determines if the server has been started and is actively listening for incoming connection requests from client applications. This method will return zero if the server is not active, if it has been suspended using the **Suspend** method or if the **Stop** method has been called to shutdown the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsActive, Start, Stop, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsLocked Method

```
BOOL IsLocked();
```

Determine if the server is currently in a locked state.

Parameters

None.

Return Value

If the server is locked, the method returns a non-zero value. If the server is not locked, the return value is zero.

Remarks

The **IsLocked** method determines if a server has been locked using the **Lock** method. Only the thread that has locked the server may interact with it and all other threads will block when they attempt to perform a network operation. After the server is unlocked, the blocked threads will resume normal execution. If the application has created multiple instances of the **CInternetServer** class, this method will return a non-zero value if any of those servers have been locked, not only the current instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Lock, Unlock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsProtocolAvailable Method

```
BOOL IsProtocolAvailable(
   INT nAddressFamily,
   INT nProtocol
);
```

The **IsProtocolAvailable** method determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the system is capable of creating a socket using the specified address family and protocol, this method will return a non-zero value. If the combination of address family and protocol is not supported, this method will return a value of zero.

Remarks

The **IsProtocolAvailable** method is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!pSocket->IsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAddress, GetHostAddress, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsReadable Method

```
BOOL IsReadable();

BOOL IsReadable(
   SOCKET hSocket
);

BOOL IsReadable(
   SOCKET hSocket,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the server can read data from the client within the specified timeout period, the method returns a non-zero value. If there is no data available to be read, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in ***lpdwAvail*** will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsWritable, Peek, Read, ReadLine, ReadStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If data can be sent to the client within the specified timeout period, the method returns a non-zero value. The method will return zero if the socket send buffer is full.

Remarks

The **IsWritable** method cannot be used to determine the amount of data that can be sent to the client without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **WriteStream** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Lock Method

```
BOOL Lock();
```

Lock the server, causing other client threads to block until it is unlocked.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Lock** method causes the specified server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This method should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this method, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this method will fail and return a value of zero.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsLocked, Unlock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::MatchHostName Method

```
BOOL MatchHostName(
   LPCTSTR lpszHostName,
   LPCTSTR lpszHostMask
   BOOL bResolve
);
```

The **MatchHostName** method matches a host name against one more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetHostAddress, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Peek Method

```
INT Peek(
   SOCKET hSocket,
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Peek(
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

Read data from the client without removing it from the socket buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

Pointer to the buffer in which the data will be stored. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the ***lpBuffer*** parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, Read, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(
    SOCKET hServer,
    UINT nClientId,
    UINT nEventId,
    DWORD dwError,
    BOOL& bHandled
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see Server Event Constants.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Read Method

```
INT Read(
   SOCKET hSocket,
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

Read data from the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

Pointer to the buffer in which the data will be stored. This parameter cannot be NULL.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes read. A return value of zero indicates that the client has closed the connection and there is no more data available to be read. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **Read** method will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this method is invoked, the session thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This method will return if any amount of data is sent by the client, and will not block until the entire buffer has been filled.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the client is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** method will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the client is sending string data to the server, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the client is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **ReadLine** method will return a line of text at a time and perform this conversion for you.

To determine if there is data available to be read without causing the session thread to block, call the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, Peek, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::ReadLine Method

```
BOOL ReadLine(
   SOCKET hSocket,
   LPTSTR lpszBuffer,
   LPINT lpnLength
);

BOOL ReadLine(
   LPTSTR lpszBuffer,
   LPINT lpnLength
);

BOOL ReadLine(
   SOCKET hSocket,
   CString& strBuffer,
   INT nMaxLength
);

BOOL ReadLine(
   CString& strBuffer,
   INT nMaxLength
);
```

Read up to a line of data from the socket and return it in a string buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the socket and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a

single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters. This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pServer->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pServer->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

IsReadable, Peek, Read, ReadStream, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::ReadStream Method

```
BOOL ReadStream(
   SOCKET hSocket,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions,
   LPBYTE lpMarker,
   DWORD cbMarker
);

BOOL ReadStream(
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions,
   LPBYTE lpMarker,
   DWORD cbMarker
);
```

Read a stream of data from the client and store it in the specified buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvBuffer

Pointer to the buffer that will contain or reference the data when the method returns. The actual argument depends on the value of the ***dwOptions*** parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the method returns. This argument should should always point to an initialized value. If the lpvBuffer argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream.

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the ***lpvBuffer*** parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the method will return a handle which references the data; otherwise, the method will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the method returns. In most cases, it is recommended that an application explicitly specify the stream

	buffer type rather than using the default value.
INET_STREAM_MEMORY	The ***lpvBuffer*** argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the method returns. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The ***lpvBuffer*** argument specifies a pointer to a global memory handle. When the method returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the method does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this method.
INET_STREAM_HANDLE	The ***lpvBuffer*** argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the **CreateFile** function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The ***lpvBuffer*** argument specifies a socket handle. The data read from the socket specified by the ***hSocket*** argument will be written to this socket. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the **AttachHandle** method or use the INET_STREAM_HANDLE stream buffer type instead.
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this method.

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the method will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **Read** method, which will return immediately when any amount of data has been read, **ReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs. This method will force the session thread to block until the operation completes.

It is possible for data to be returned in the buffer even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Example

```
HGLOBAL hgblBuffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400;   // Read up to 100K bytes
BOOL bResult;

bResult = pServer->ReadStream(&hgblBuffer, &cbBuffer,
                              INET_STREAM_HGLOBAL | INET_STREAM_CONVERT);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Use data in the stream buffer

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetStreamInfo, Read, ReadLine, StoreStream, Write, WriteLine, WriteStream

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Reject Method

```
BOOL Reject();
```

Reject a pending client connection.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the server thread to block. This method should only be invoked from within an **OnAccept** event handler if the application wishes to reject the incoming connection before a client session is created.

To determine the IP address of a client that is attempting to connect to the server from within the **OnAccept** event, use the **GetClientAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientAddress, OnAccept

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Restart Method

```
BOOL Restart();
```

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the options specified for the previous instance.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Start, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Resume Method

```
BOOL Resume();
```

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Restart, Start, Stop, Suspend, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetBacklog Method

```
BOOL SetBacklog(
   UINT nBacklog
);
```

Set the size of the backlog connection queue for the server.

Parameters

nBacklog

An integer value that specifies the size of the connection backlog queue.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **SetBacklog** method specifies the size of the queue allocated for pending client connections. A value of zero specifies that the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. This method should only be called prior to invoking the **Start** method, it does not have any effect on an active server. To get the current size of the backlog queue, use the **GetBacklog** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetBacklog, Start, Data Members

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetCertificate Method

```
BOOL SetCertificate(
   DWORD dwProtocol,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);

BOOL SetCertificate(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);

BOOL SetCertificate(
   LPCTSTR lpszCertName,
   LPCTSTR lpszPassword
);
```

Specify the server certificate that should be used with secure connections.

Parameters

dwProtocol

An optional bitmask of supported security protocols. If this parameter is not specified, then a default set of security protocols will be automatically selected. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default selection of security protocols will be used when establishing a connection. The TLS 1.2, TLS 1.1 and TLS 1.0 protocols will be negotiated with the client, in that order of preference. This option will always request the latest version of the preferred security protocols and is the recommended value.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the client. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Note that SSL 2.0 has been deprecated and will never be used by default.
SECURITY_PROTOCOL_TLS	The TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the client. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some clients that reject any attempt to use the older SSL protocol and require that only TLS be used.

lpszCertStore

An optional string value that specifies the name of the certificate store that contains the server certificate. If the certificate is stored in the registry, this parameter must specify a valid local certificate store name. If the certificate is stored in a file, this parameter should specify the full path to the file that contains the certificate. If this parameter is omitted, the personal certificate store for the current process will be used.

lpszCertName

A string value that specifies the name of the certificate. This parameter is required and cannot be NULL. Either the common name or the name assigned to the certificate may be specified. In most cases, this will be the fully qualified domain name of the host that the server is running on.

lpszPassword

An optional string value that specifies a password associated with the server certificate. This parameter is usually only required when the ***lpszCertStore*** parameter specifies a certificate stored in a file. If the server certificate does not have a password associated with it, this parameter or omitted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableSecurity, SetOptions, Start, ValidateCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetClientData Method

```
BOOL SetClientData(
   SOCKET hClient,
   LPVOID lpvData
);

BOOL SetClientData(
   LPVOID lpvData
);
```

Associate application defined data with the specified client session.

Parameters

hClient

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvData

Pointer to the application defined data associated with the specified client session.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be modified. To get extended error information, call the **GetLastError** method.

Remarks

The **SetClientData** method is used to associate application defined data with a specific client session. This is typically used to associate a pointer to a data structure or a class instance with the client socket. The data can be retrieved using the **GetClientData** method.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling method exits. If you do this, the pointer will no longer be valid after the method exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a method such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization methods when dereferencing and modifying that structure.

Example

```
UINT *pnValue1 = new UINT;
UINT *pnValue2 = NULL;

*pnValue1 = 1234;

if (!pServer->SetClientData(hClient, pnValue1))
{
    // Unable to associate the data with this session
    return;
}

if (!pServer->GetClientData(hClient, (LPVOID *)&pnValue2))
{
    // Unable to retrieve the data associated with this session
```

```
    return;
}

// pnValue2 == pnValue1
printf("The value of the user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetClientData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetClientMoniker Method

```
INT SetClientMoniker(
   SOCKET hSocket,
   LPCTSTR lpszMoniker
);

INT SetClientMoniker(
   LPCTSTR lpszMoniker
);
```

Associate a unique string alias with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszMoniker

Pointer to a string which specifies the moniker for the specified client socket. If this parameter is NULL or specifies an empty string, a moniker will no longer be associated with the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. The **GetClientMoniker** method will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **FindClient** method.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

FindClient, GetClientHandle, GetClientId, GetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetOptions Method

```
BOOL SetOptions(
   DWORD dwOptions
);
```

Set one or more server options.

Parameters

dwOptions

An unsigned integer that specifies one or more option bit flags.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **SetOptions** method sets the server options for the class instance. For a list of the available options, see Server Option Constants. This method should only be called prior to invoking the **Start** method, it does not have any effect on an active server. The **GetOptions** method returns the options that are currently specified for the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetOptions, Data Members

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetPriority Method

```
INT SetPriority(
   INT nPriority
);
```

Set the priority assigned to the specified server.

Parameters

nPriority

An integer that specifies the server priority. It may be one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is INET_ERROR.

Remarks

The **SetPriority** method changes the current priority assigned to the specified server. Client connections that are accepted after this method is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetStackSize Method

```
BOOL SetStackSize(
   DWORD dwStackSize
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change this value unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetStackSize, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

Set the timeout interval used when waiting for a blocking operation to complete.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetTimeout** method sets the amount of time that the server will wait for data to become available to read, and the default timeout for blocking network operations for each client session. If this method is invoked before the server has started, it will change the default timeout period for the entire server. If this method is invoked within a server event handler, it will change the timeout period for the active client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetTimeout, IsReadable, IsWritable, OnTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

Display a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Remarks

The **ShowError** method will display a modal message box with an error message that corresponds to the specified error code. All top-level windows belonging to the current thread will be disabled until the user responds to the message box. An application should only invoke this method from within the main UI thread, never from within a server event handler such as **OnConnect**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Start Method

```
BOOL Start(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   UINT nMaxClients,
   DWORD dwOptions
);

BOOL Start(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   DWORD dwOptions
);

BOOL Start(
   UINT nLocalPort,
   DWORD dwOptions
);

BOOL Start(
   UINT nLocalPort
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

nMaxClients

The maximum number of client connections that can be established with the server. A value of zero specifies that there should not be any fixed limit on the number of active client connections. This value can be adjusted after the server has been created by calling the **Throttle** method.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see Server Option Constants. If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call **GetLastError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The server instance is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

Example

```
// EchoServer implementation
class CEchoServer : public CInternetServer
{
    void OnRead(SOCKET hSocket)
    {
        // Read data sent by the client to the server
        BYTE ioBuffer[1024];
        INT nBytesRead = Read(hSocket, ioBuffer, sizeof(ioBuffer));

        // Send a copy of the data back to the client
        if (nBytesRead > 0)
            Write(hSocket, ioBuffer, nBytesRead);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    CEchoServer myServer;

    // Start the server listening for connections on port 7000
    if (myServer.Start(7000))
    {
        TCHAR szCommand[128], *pszCommand;

        // Process commands entered by the user at the console
        while (TRUE)
        {
            if ((pszCommand = _fgetts(szCommand, 128, stdin)) == NULL)
                break;

            if (_tcsicmp(pszCommand, _T("quit")) == 0)
                break;
        }
```

```
        // Stop the server and terminate all clients
        myServer.Stop();
    }

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumClients, Restart, Stop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Stop Method

```
BOOL Stop();
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Restart, Start

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::StoreStream Method

```
BOOL StoreStream(
   LPCTSTR lpszFileName,
   DWORD dwLength,
   LPDWORD lpdwCopied
   DWORD dwOffset,
   DWORD dwOptions
);
```

The **StoreStream** method reads the socket data stream and stores the contents in the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the method will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the method returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Value	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the

	source data stream. If the *dwOffset* parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than memory buffers or handles.

Example

```
// Store all data sent by the client in a file
DWORD dwCopied = 0;
BOOL bResult = pServer->StoreStream(lpszFileName, 0, &dwCopied, 0,
INET_STREAM_CONVERT);

// Close the client connection to the server
pServer->Disconnect();

if (bResult && dwCopied > 0)
{
    // Process the data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Read, ReadLine, ReadStream, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Suspend Method

```
BOOL Suspend();
```

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Restart, Resume, Start, Stop, Throttle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Throttle Method

```
BOOL Throttle(
   UINT nMaxClients,
   UINT nMaxClientsPerAddress,
   DWORD dwConnectionRate
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of

time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Restart, Resume, Start, Suspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Unlock Method

```
BOOL Unlock();
```

Unlock the server, allowing other client threads to resume execution.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Unlock** method releases the lock on the specified server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this method, and should never assume that the lock has been released. If a potential deadlock situation is detected, this method will fail and return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsLocked, Lock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::ValidateCertificate Method

```
BOOL ValidateCertificate(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertPassword,
   LPCTSTR lpszCertName
);
```

The **ValidateCertificate** method determines if the specified security certificate is installed on the local system.

Parameters

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the personal certificate store will be used as the default. This parameter may also specify the name of a certificate file in PKCS #12 (PFX) format.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertPassword

A null-terminated string which specifies the password associated with a certificate file. This parameter is only used if the ***lpszCertStore*** parameter specifies a certificate file, otherwise it is ignored. If the certificate file is not protected with a password, this parameter should be a NULL pointer or empty string.

lpszCertName

A null-terminated string which specifies the name of the certificate to validate. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call **InetGetLastError**.

Remarks

If you are checking the validity of a certificate installed in the local certificate store, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user.

It is possible to validate a certificate file rather than one stored in the local certificate store. The *lpszCertStore* member should specify the name of a file in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate file was created, it must be provided in with the *lpszCertPassword* parameter or this method will be unable to access the certificate.

This method can only validate certificate files in PFX format and cannot be used to validate a certificate file in another format, such as PEM (base64 encoded) or DER (binary).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableSecurity, SetCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::Write Method

```
INT Write(
   SOCKET hSocket,
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

Send the contents of the specified buffer to to the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the client.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, Read, ReadLine, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::WriteLine Method

```
BOOL WriteLine(
   SOCKET hSocket,
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);

BOOL WriteLine(
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

Write a line of data to the client, terminated with a carriage-return and linefeed.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, the **WriteLine** method is specifically designed to write a single line of text data from a null-terminated string. This method will force the session thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** method, it is possible for data to have been written to the socket if the return value is

zero. For example, if a timeout occurs while the method is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsWritable, Read, ReadLine, ReadStream, Write, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::WriteStream Method

```
BOOL WriteStream(
   SOCKET hSocket,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);

BOOL WriteStream(
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);
```

Write a stream of data to the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the ***dwOptions*** parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the method returns. This argument should should always point to an initialized value. If the ***lpvBuffer*** argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the ***lpvBuffer*** parameter. If the argument specifies a a global memory handle, then the method will write the data referenced by that handle; otherwise, the method will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The ***lpvBuffer*** argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned

	integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The ***lpvBuffer*** argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The ***lpvBuffer*** argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The ***lpvBuffer*** argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the ***hSocket*** parameter. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **Write** method, which may not write all of the data in a single method call, **WriteStream** will only return when all of the data has been written or an error occurs. This method will force the thread to block until the operation completes.

It is possible for some data to have been written even if the method returns a value of zero. Applications should also check the value of the ***lpdwLength*** argument to determine if any data was sent. For example, if a timeout occurs while the method is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Example

```
CFile *pFile = new CFile();
DWORD dwLength = 0;

if (!pFile->Open(strFileName, CFile::modeRead | CFile::shareDenyWrite))
    return;

dwLength = pFile->GetLength();
```

```
if (dwLength > 0)
{
    BOOL bResult = pServer->WriteStream(
                      pFile->m_hFile,
                      &dwLength,
                      INET_STREAM_HANDLE);
}

delete pFile;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Read, ReadLine, ReadStream, StoreStream, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer Event Handlers

Method	Description
OnAccept	The client is attempting to establish a connection to the server
OnConnect	The client has established a connection to the server
OnDisconnect	The client has disconnected from the server
OnError	The event handler encountered an error when processing a client event
OnRead	The client has sent data to the server
OnTimeout	The client has not sent data within the specified timeout period
OnWrite	The client is ready to receive data from the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnAccept Method

```
virtual void OnAccept(
   SOCKET hSocket
);
```

A virtual method that is invoked when a client attempts to connect to the server.

Parameters

hSocket

A handle to the server socket.

Return Value

None.

Remarks

The **OnAccept** event handler is invoked when a client attempts to connect to the server, but prior to the connection being accepted. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event only occurs before the server has checked the active client limits. This event is typically used to reject a connection based on some criteria established by the server, such as the IP address of the client attempting to make the connection. To obtain the IP address of the client that is attempting to connect to the server, use the **GetClientAddress** method using the server handle.

If this event handler is not implemented, the server will permit the client connection to complete. To reject the connection attempt, call the **Reject** method using the handle to the server socket. Rejecting the client connection within the **OnAccept** event handler may cause unexpected behavior by the client application because the connection process will not complete normally. Instead of rejecting the client connection within the **OnAccept** handler, it is recommended that most server applications implement an **OnConnect** event handler, perform any required checks and then gracefully disconnect the client using the **Disconnect** method if needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, OnConnect, OnDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnConnect Method

```
virtual void OnConnect(
   SOCKET hSocket,
   UINT nClientId,
   LPCTSTR lpszAddress,
   UINT nPort
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

hSocket

A handle to the client socket.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

nPort

An integer that specifies the port number that the client socket is bound to.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and the TLS handshake has been performed, if security has been enabled. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will reject the connection prior to this event handler being invoked.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, OnAccept, OnDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnDisconnect Method

```
virtual void OnDisconnect(
   SOCKET hSocket
);
```

A virtual method that is invoked when a client disconnects from the server.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked when a client disconnects from the server, immediately before the client session is terminated. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, OnConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnError Method

```
virtual void OnConnect(
   SOCKET hSocket,
   UINT nEventId,
   DWORD dwError
);
```

A virtual method that is invoked when the server encounters an error while handling a client request.

Parameters

hSocket

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see Server Event Constants.

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit TLS but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- Network errors that may occur when attempting to buffer data sent by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

OnConnect, OnDisconnect, OnTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnRead Method

```
virtual void OnRead(
   SOCKET hSocket
);
```

A virtual method that is invoked when a client sends data to the server.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnRead** event handler is invoked when a client sends data to the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method. All server applications must implement an **OnRead** event handler to process the data sent by the client.

This event occurs whenever there is data available to be read from the client. The server application reads the data using the **Read** or **ReadLine** method, and can then send data back to the client using the **Write** or **WriteLine** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Read, ReadLine, Write, WriteLine, OnWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnTimeout Method

```
virtual void OnTimeout(
   SOCKET hSocket
);
```

A virtual method that is invoked when the client has not sent any data to the server within the timeout period.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked when a client has not sent any data to the server within the timeout period specified when the server was started. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event handler is typically used to monitor the amount of time that a client is idle. The default timeout period for the server is 20 seconds, which would cause this event handler to be invoked whenever a client has not sent any data to the server in the last 20 seconds. The server may take no action, or it may disconnect the client after it has been idle for an extended period of time. To get the total amount of time that the client has been idle, call the **GetClientIdleTime** method. Note that while the server timeout period is specified in seconds, the **GetClientIdleTime** method returns the client idle time in milliseconds.

The default implementation for this event handler is to take no action. It is recommended that most server applications disconnect clients that are inactive. For typical client-server implementations that use transitory connections (where the client sends a single request to the server, the server responds and the connection is terminated) the amount of time that a client should be permitted to remain idle should be relatively low, usually 60 seconds or less. For persistent connections where there are multiple requests issued by the client over the lifetime of the session, a longer idle timeout period may be preferable.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CInternetServer::OnWrite Method

```
virtual void OnWrite(
   SOCKET hSocket
);
```

A virtual method that is invoked when the client is ready to receive data.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnWrite** event handler is invoked when a client is ready to receive data. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method. All server applications must implement an **OnRead** event handler to process the data sent by the client.

This event occurs immediately after the **OnConnect** event and if security is enabled, after the TLS handshake has completed. It is used to notify the server application that the client is ready to receive data, and may be used to send an initial message to the client, typically identifying the server that it has connected to. In most cases, the **OnWrite** event handler will only be invoked once over the lifetime of the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Write, WriteLine, OnRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Server Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO

This structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD   dwStreamThread;
    DWORD   dwStreamSize;
    DWORD   dwStreamCopied;
    DWORD   dwStreamMode;
    DWORD   dwStreamError;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Value	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

See Also

ReadStream, StoreStream, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
     INT    ipFamily;
     BYTE   ipNumber[16];
}  INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient Class Library

The **CLlmClient** class provides a unified, high-level interface for integrating large language model capabilities into applications. It simplifies working with different providers by handling connections, messaging, and conversation state internally, while still giving developers control over configuration, performance, and behavior. The class is designed to support both simple use cases and more advanced scenarios involving model selection, token management, and session control.

Reference

- Class Methods
- Constants
- Data Structures
- Providers
- Error Codes

Library Information

Class Name	CLlmClient
File Name	CSLLMV12.DLL
Version	12.0.3024.1286
LibID	E77A6A0A-3B86-49D9-9A3D-A0B425BDA66D
Import Library	CSLLMV12.LIB
Dependencies	None

Overview

The **CLlmClient** class is built around a session-based architecture that encapsulates all aspects of interacting with a language model, including connection details, configuration settings, and conversation history. This design allows developers to treat each interaction as part of a managed context, making it easier to maintain state across multiple requests without needing to manually track prior messages or model parameters. Sessions can be configured with defaults or customized with options such as system prompts, token limits, and behavioral settings, depending on the needs of the application.

A central goal of the class is to abstract the differences between providers and models, presenting a unified interface that works consistently regardless of the underlying service. Developers can discover available providers and models, query their capabilities, and select appropriate configurations at runtime. This flexibility makes it possible to switch between services or adapt to new models with minimal code changes, while still exposing enough detail to fine-tune behavior when needed.

The class also provides robust support for managing conversation data. It maintains a structured history of messages and responses, enabling applications to iterate through prior interactions, search for specific content, or limit the amount of retained context to control memory usage and token consumption. This is particularly important when working with models that have strict context limits, as it allows developers to balance response quality with performance and cost considerations.

In addition to core messaging functionality, the class includes a range of utility features that help developers monitor and optimize their applications. These include estimating and tracking token

usage, managing cache lifetimes for model metadata, handling errors in a consistent way, and optionally logging network activity for debugging purposes. Together, these capabilities provide insight into how the application is interacting with the model and help identify potential performance or configuration issues.

Overall, the **CLlmClient** class is designed to provide a practical balance between simplicity and control. It enables developers to quickly add AI-powered features to their applications while retaining the flexibility to customize behavior, manage resources, and integrate with different providers as requirements evolve.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the method declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient Methods

Class	Description
CLlmClient	Constructor which initializes the current instance of the class.
~CLlmClient	Destructor which releases resources allocated by the class.
Method	Description
AskModel	Sends a prompt to the specified service provider and model.
AttachHandle	Attaches the specified client handle to this instance of the class.
AttachThread	Attaches the specified client handle to another thread.
Cancel	Cancels the current blocking request.
ClearHeaders	Clears the custom HTTP request headers for the client session.
ClearHistory	Clears the current conversation history for the client.
Connect	Establishes a connection using the specified provider, URL, model, API key, and options.
ConnectEx	Establishes a connection using an existing session structure.
ConvertTimestamp	Converts a message or response timestamp value into a SYSTEMTIME structure.
CreateSession	Creates an internal session object using the specified provider settings.
DeleteSession	Deletes an internal session object and releases its resources.
DetachHandle	Detaches the handle for the current instance of this class.
DisableTrace	Disables logging of network function calls.
Disconnect	Disconnects from the provider and releases the session handle.
EnableTrace	Enables logging of network function calls to a file.
EnumModels	Enumerates the models available for the connected provider.
EnumProviders	Enumerates the supported LLM providers.
EstimateTokenCount	Estimates the number of tokens in a text string for a specific provider.
ExportHistory	Exports the current conversation history to a JSON-formatted file.
FindMessage	Searches the conversation history for a message matching the specified text or flags.
FormatTimestamp	Formats a message timestamp as a ISO 8601 date and time string.
GetCacheTime	Returns the current model or metadata cache lifetime for the client.
GetDefaultModelFlags	Returns the default capability flags for the specified model.
GetDefaultModelName	Returns the default model name for the specified provider.
GetDefaultUrl	Returns the default endpoint URL for the specified provider.
GetErrorString	Returns a text description for the specified error code.
GetFirstMessage	Retrieves the first message in the conversation history.
GetHandle	Returns the client handle used by this instance of the class.
GetHeader	Returns the value of an HTTP header from the server response.
GetHistorySize	Returns the maximum number of messages which can be stored in the conversation history.
GetIdleTime	Returns the current idle timeout period for the client.
GetLastError	Returns the last error code for the current thread.

GetMemoryUsage	Returns the amount of memory currently used for the client session.
GetMessage	Retrieves a specific message prompt and response using the history index.
GetMessageById	Retrieves a specific message prompt and response pair by message ID.
GetMessageCount	Returns the current number of messages in the conversation history.
GetModelContextSize	Returns the context window size for the specified model.
GetModelFlags	Returns the current model behavior flags for the client.
GetModelInfo	Returns detailed information about a specific model.
GetModelName	Returns the name of the current model used with a client session.
GetNextMessage	Retrieves the next message in the conversation history.
GetOptions	Returns the current client option flags.
GetProviderUrl	Returns the provider base URL used for client requests.
GetProviderFlags	Returns the capability flags associated with a provider.
GetProviderId	Returns the current provider ID or one based on a provider name, endpoint URL or model name.
GetProviderInfo	Retrieves detailed information about a provider.
GetProviderName	Returns the display name of a provider.
GetProviderUrl	Returns the provider endpoint URL for the current session.
GetSession	Returns a pointer to the current LLM_SESSION structure.
GetStatusCode	Returns the most recent HTTP status code for the client request.
GetSystemPrompt	Returns the current system prompt for the client session.
GetTimeout	Returns the number of seconds until a request times out.
GetTokenUsage	Returns token usage statistics for the client session.
GetUserAgent	Returns the current user-agent string used by the client.
ImportHistory	Imports a conversation history from the specified JSON-formatted file.
IsBlocking	Returns whether the client is currently processing a blocking operation.
IsConnected	Returns whether the client session is valid and connected to a server.
IsInitialized	Determines if the class has been successfully initialized.
NormalizeText	Normalizes message text into a consistent format.
ResetSession	Resets the current client session to its initial state.
SendMessage	Sends a prompt to the model and returns the response as a string.
SendMessageEx	Sends a structured message to the model and returns a structured response.
SetCacheTime	Sets the cache lifetime used by the client for model or provider data.
SetHeader	Sets the value of a custom HTTP header issued with subsequent requests.
SetHistorySize	Sets the maximum number of messages retained in the conversation history.
SetIdleTime	Sets the idle timeout period for the client session.
SetLastError	Sets the last error code for the current thread.
SetModelFlags	Sets model behavior flags for the client session.
SetModelName	Changes the current model for a client session.
SetOptions	Sets the current client option flags.
SetSystemPrompt	Sets the system prompt for the client session.

SetTimeout	Sets the number of seconds until a blocking operation times out.
SetUserAgent	Sets the user-agent string used by the client.
ShowError	Displays a message box with a description of the specified error.
TrimHistory	Removes older messages from the conversation history, optionally constrained by token usage.
ValidateModel	Validates that a model name is recognized or supported by the current provider.
ValidateText	Performs basic validation on message text before it is submitted to a model.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::~CLlmClient

```
~CLlmClient();
```

The **CLlmClient** destructor releases resources allocated by the current instance of the **CLlmClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CLlmClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

CLlmClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class methods may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the thread that created it. If another thread attempts to use the handle, the method will fail because that thread does not own the handle. This prevents other threads from interfering with an operation being performed by the owner thread. The **AttachThread** method changes the thread that owns the client handle. This is useful when one thread creates the client session and then passes it to a worker thread for processing. The method returns the thread ID of the previous owner, which can be saved and used later to restore ownership of the handle.

This method should normally be called by the new owner thread immediately after it has been created. If the worker thread does not release the handle itself, ownership should be restored to the original thread before the worker thread terminates. Do not use **AttachThread** to forcibly take ownership of a handle from another thread while a blocking operation is in progress. The handle should only be transferred when no other thread is actively using it.

The ***dwThreadId*** parameter is assumed to be a valid thread ID. This method does not verify that the specified thread exists. Specifying an invalid thread ID will orphan the handle until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Connect, Disconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::Cancel Method

```
BOOL Cancel();
```

The **Cancel** method cancels the current blocking request.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Cancel** method cancels the current blocking request by closing the active HTTP connection associated with the client session. This causes the blocking operation to fail and return control to the calling thread. The last error code for the blocking function will be set to indicate the operation was canceled.

This method cancels the request locally. It does not guarantee that the provider will stop processing a request that has already been received. The provider may still complete the request and may charge for any processing or tokens used before the connection was closed.

Canceling a request does not delete the client session or clear the conversation history. The client handle remains valid, and a new HTTP connection will be established automatically if it is needed for a subsequent request.

If no blocking request is currently in progress, this method succeeds and has no effect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

IsBlocking, SendMessage, SendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ClearHeaders Method

```
BOOL ClearHeaders();
```

The **ClearHeaders** method removes all custom HTTP request headers for the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ClearHeaders** method removes all custom request header values set using the **SetHeader** method. This is useful when you want to ensure that custom headers used for a previous call to **SendMessage** are not included with subsequent requests made during the same session.

This method does not clear authentication credentials, organization identifiers, project identifiers, or any required headers automatically generated by the control.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetHeader, ResetSession, SendMessage, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ClearHistory Method

```
BOOL ClearHistory();
```

The **ClearHistory** method clears the conversation history for the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ClearHistory** method removes all messages and responses stored in the conversation history for the specified client session. This can be used to begin a new conversation while preserving the current connection, provider settings, model selection, timeout, system prompt and other session options.

This method only clears the local history maintained by the client. It does not reset provider-specific settings, delete cached model information or close the active HTTP connection. It also does not change the current system prompt.

After the history has been cleared, subsequent calls to **SendMessage** or **SendMessageEx** will not include any previous messages from the conversation history.

Clearing the conversation history invalidates any string pointers previously returned in LLM_RESPONSE structures or by message history methods. Applications must not use those pointers after this method returns. If the application needs to keep any message or response text, it should make its own copy before clearing the history.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetHistorySize, SetHistorySize, GetFirstMessage, GetNextMessage, ResetSession, TrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::CLlmClient

```
CLlmClient();
```

The **CLlmClient** constructor initializes the class library and validates the runtime license key.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **LlmInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

~CLlmClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::Connect Method

```
BOOL Connect(
   DWORD dwProviderId,
   DWORD dwOptions,
   INT nTimeout,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey
);
```

The **Connect** method establishes a connection to the specified large language model service provider and creates a client session.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. See the remarks below for more information about service providers.

dwOptions

Specifies one or more option flags that control how the connection is established. This parameter can be zero or a combination of the following values:

Value	Description
LLM_OPTION_NONE	A standard connection to the server is established using the default behavior for the selected provider. No additional options are enabled, and the connection will use the provider's default protocol, caching behavior and request headers.
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be

	used when a provider does not require a User-Agent header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.
LLM_OPTION_DEFAULT	Specifies the default option set, which is equivalent to `LLM_OPTION_NONE`. No additional flags are enabled and the connection uses standard behavior.

nTimeout

The timeout period for connection and request operations, in seconds. A value of zero indicates that the default timeout should be used.

lpszBaseUrl

A pointer to a null-terminated string that specifies the base URL for the provider endpoint. This parameter may be NULL to use the default endpoint for the selected provider. Some providers, such as Microsoft Foundry or local services, require a specific base URL.

lpszModelName

A pointer to a null-terminated string that specifies the model name to use for the session. This parameter may be NULL to use the provider default model, if one is defined. Providers without a default model require an explicit model name.

lpszApiKey

A pointer to a null-terminated string that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Connect** method creates a client session that is used for all subsequent operations with the selected provider. The returned handle maintains the connection settings, authentication information, selected model, and conversation history for the session.

The *dwProviderId* parameter specifies the numeric identifier assigned to a supported provider. Provider identifiers are predefined constants built into the library and are assigned fixed values beginning at 1 and increasing sequentially for each provider. These identifiers are stable and are intended to uniquely identify a provider regardless of its display name. Applications should use the predefined constants (such as `LLM_PROVIDER_OPENAI` or `LLM_PROVIDER_MICROSOFT`) rather than hard-coding numeric values directly in source code.

Refer to the providers page for a list of supported service providers and their default models.

If the *lpszBaseUrl* parmeter is NULL or a zero-length string, the library will use a provider-specific default endpoint. Some providers, such as Microsoft Foundry services or locally hosted models,

require an explicit endpoint and will fail if one is not provided.

If the *lpszModelName* parameter is NULL or a zero-length string, the library will attempt to use a default model for the selected provider. Not all providers define a default model, and in those cases an explicit model name must be provided when establishing the connection. This typically includes models accessed using the Microsoft Foundry or locally hosted inference providers.

If the service provider requires authentication, the API key can be provided in several formats, allowing applications to provide credentials in a way that balances convenience and security:

- **Literal string**
  The API key is provided directly as a string. This is the simplest approach and is convenient for testing or small internal tools, but it is not recommended for production applications. A literal key may be exposed in source code, configuration files, diagnostic logs or the compiled application binary.
- **Environment variable**
  The name of an environment variable enclosed in percent symbols, for example `"%OPENAI_API_KEY%"`. The value of the environment variable is used as the API key. This avoids embedding the key directly in the application and is useful for development, scripts and deployment environments. However, environment variables may still be visible to the current user, inherited by child processes or exposed through system configuration, so they should still be protected appropriately.
- **Windows credential**
  A credential name enclosed in braces, for example `"{MyAppName/OpenAI}"`. The API key is retrieved from the Windows Credential Manager using the specified name. This is the recommended approach for most Windows applications because the secret is stored outside the application and managed by the operating system. The credential can be created manually using Credential Manager, or programmatically by the application using the Windows Credential Management API. Applications should choose a unique credential name and document it clearly if users or administrators are expected to configure the value themselves.

For production applications, it is recommended that you avoid using literal API key strings and instead use an environment variable or a value stored in the Windows Credential Manager. If you must provide the API key as a literal string, you should take additional steps to protect it, such as storing the key in an encrypted form and limiting access to authorized users or processes. Avoid embedding API keys directly in application source code, configuration files, installers or other content that could be exposed through version control systems, logs or public repositories.

This method will block the calling thread until the connection has been established or the timeout period has elapsed. The connection established by this method represents a logical session. Depending on the provider and options specified, the underlying HTTP connection may be reused across requests or recreated as needed. Options such as `LLM_OPTION_KEEPALIVE` can be used to control connection behavior.

If the class instance is connected to another provider, this method will explicitly close that connection by calling the **Disconnect** method prior to establishing the new connection. If the connection fails, the **GetHandle** method will return INVALID_HANDLE.

The client handle returned by this method is owned by the thread that created it. To transfer ownership to another thread, use the **AttachThread** method.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    CLlmClient llmClient;

    // Establish a connection to OpenAI using the default endpoint
    // and a lightweight model
    BOOL bConnected = llmClient.Connect(
        LLM_PROVIDER_OPENAI,
        LLM_OPTION_DEFAULT,
        LLM_TIMEOUT,
        NULL,                   // Default endpoint
        _T("gpt-5.4-nano"),     // Model name
        _T("%OPENAI_API_KEY%") // Environment variable
    );

    if (!bConnected)
    {
        DWORD dwError = llmClient.GetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        return 1;
    }

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse); // number of TCHARs
    DWORD dwMessageId = 0;

    // Send a simple message to the model
    if (llmClient.SendMessage(
            _T("What is the capital city of Denmark?"),
            szResponse,
            &dwResponseLength,
            &dwMessageId))
    {
        _tprintf(_T("The response was %lu characters:\n"), dwResponseLength);
        _putts(szResponse);
    }
    else
    {
        DWORD dwError = llmClient.GetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

     // Close the connection explicitly
    llmClient.Disconnect();
    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectEx, Disconnect, EnumModels, GetProviderInfo, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ConnectEx Method

```
BOOL ConnectEx(
   DWORD dwProviderId,
   DWORD dwOptions,
   LPLLM_SESSION lpSession
);

BOOL ConnectEx();
```

The **ConnectEx** method establishes a connection to the specified large language model service provider using an extended session configuration.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. See the remarks below for more information about service providers.

dwOptions

Specifies one or more option flags that control how the connection is established. This parameter can be zero or a combination of **LLM_OPTION** values. These options control behavior such as connection security, proxy usage, and HTTP connection handling.

Value	Description
LLM_OPTION_NONE	A standard connection to the server is established using the default behavior for the selected provider. No additional options are enabled, and the connection will use the provider's default protocol, caching behavior and request headers.
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be

	used when a provider does not require a User-Agent header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.
LLM_OPTION_DEFAULT	Specifies the default option set, which is equivalent to `LLM_OPTION_NONE`. No additional flags are enabled and the connection uses standard behavior.

lpSession

A pointer to an optional `LLM_SESSION` structure that specifies extended configuration options for the client session. This structure can be used to define parameters such as the base URL, model name, authentication information, timeout values, token limits and sampling settings. If this parameter is omitted, or is NULL, the method will use session information from a previous call to **CreateSession**.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ConnectEx** method creates a client session using the configuration specified in the *lpSession* structure. This provides more control over how the connection is established compared to **Connect**, and is intended for applications that require advanced configuration.

A version of this method can be called without any parameters. In this case, it will use default options and the session information specified with a previous call to the **CreateSession** method.

The *dwProviderId* parameter specifies the numeric identifier assigned to a supported provider. Provider identifiers are predefined constants built into the library and are assigned fixed values beginning at 1 and increasing sequentially for each provider. These identifiers are stable and are intended to uniquely identify a provider regardless of its display name. Applications should use the predefined constants (such as `LLM_PROVIDER_OPENAI` or `LLM_PROVIDER_MICROSOFT`) rather than hard-coding numeric values directly in source code.

Refer to the providers page for a list of supported service providers and their default models.

The *lpSession* parameter allows the application to specify connection parameters such as the provider endpoint, model name, API key, timeout values, token limits and sampling options. Any values not explicitly specified will use provider defaults. If this parameter is NULL, the method will use the internal LLM_SESSION structure allocated using the **CreateSession** method. If the *lpSession* parameter is NULL, and you have not called **CreateSession**, the method will fail.

The library makes an internal copy of the string members in the LLM_SESSION structure when the connection is established. The application does not need to retain them after the call to **ConnectEx** returns.

This method will block the calling thread until the connection has been established or the timeout period has elapsed. Applications should avoid calling this method on the main user interface thread. For best results, create a worker thread to establish the connection, especially when multiple

connections are required.

The connection established by this method represents a logical session. Depending on the provider and options specified, the underlying HTTP connection may be reused across requests or recreated as needed. Options such as `LLM_OPTION_KEEPALIVE` can be used to control connection behavior.

If the class instance is connected to another provider, this method will explicitly close that connection by calling the **Disconnect** method prior to establishing the new connection. If the connection fails, the **GetHandle** method will return INVALID_HANDLE.

The client handle returned by this method is owned by the thread that created it. To transfer ownership to another thread, use the **AttachThread** method.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    CLlmClient llmClient;

    // Provide basic session information
    // Additional options such as temperature, token limits and timeouts
    // can be configured using members of the LLM_SESSION structure
    LLM_SESSION sessionInfo;

    ZeroMemory(&sessionInfo, sizeof(sessionInfo));
    sessionInfo.dwSize = sizeof(sessionInfo);
    sessionInfo.lpszModelName = _T("gpt-5.4-nano");
    sessionInfo.lpszApiKey = _T("%OPENAI_API_KEY%");
    sessionInfo.lpszSystemPrompt = _T("You are a helpful assistant.");

    // Establish a connection to OpenAI using a lightweight model
    // with the default connection options
    BOOL bConnected = llmClient.ConnectEx(
                          LLM_PROVIDER_OPENAI,
                          LLM_OPTION_DEFAULT,
                          &sessionInfo);

    if (!bConnected)
    {
        DWORD dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        return 1;
    }

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse); // number of TCHARs
    DWORD dwMessageId = 0;

    // Send a simple message to the model
    if (llmClient.SendMessage(
                      _T("What is the capital city of Denmark?"),
                      szResponse,
                      &dwResponseLength,
                      &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
```

```
    }
    else
    {
        dwError = llmClient.GetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Close the connection explicitly
    llmClient.Disconnect();
    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSession, DeleteSession, Disconnect, EnumModels, GetProviderInfo, IsConnected, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ConvertTimestamp Method

```
BOOL ConvertTimestamp(
   TIME_T tTimestamp,
   LPSYSTEMTIME lpSystemTime
   BOOL bLocalTime
);
```

The **ConvertTimestamp** method converts a message or response timestamp to a SYSTEMTIME structure.

Parameters

tTimestamp

A 64-bit timestamp value associated with a message or response. This value represents the number of seconds that have elapsed since January 1, 1970 UTC.

lpSystemTime

A pointer to a `SYSTEMTIME` structure that receives the converted date and time value.

bLocalTime

If this parameter is non-zero, the timestamp is converted to the local system time. If this parameter is zero, the timestamp is converted to Coordinated Universal Time (UTC).

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ConvertTimestamp** method converts an internal timestamp value used by the library into a standard `SYSTEMTIME` structure. This allows applications to display message and response times in a human-readable format. These timestamps are used in the client history of requests sent to the model and the responses it returned.

The *tTimestamp* parameter is a standard 64-bit Unix time value and can be used interchangeably with functions in the C standard library such as **gmtime** or **localtime**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps are returned by functions such as **LlmSendMessage** and can be converted into a local date and time using the **LlmConvertTimestamp** function.

The converted time can be returned as either local time or UTC by specifying the *bLocalTime* parameter. Applications should use local time for display purposes and UTC when storing or comparing timestamps across systems. The SYSTEMTIME value can be formatted using standard Windows functions such as **GetDateFormatEx** and **GetTimeFormatEx** to display locale-specific date and time values.

If you wish to convert the message timestamp to a standardized ISO 8601 string value, use the **FormatTimestamp** method.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };
```

```
if (llmClient.GetFirstMessage(&clientMessage, &serverResponse))
{
    do
    {
        SYSTEMTIME stMessage = {0};
        TCHAR szDate[64] = {0};
        TCHAR szTime[64] = {0};

        llmClient.ConvertTimestamp(clientMessage.tTimestamp, &stMessage, TRUE);

        GetDateFormat(LOCALE_USER_DEFAULT, 0,
                      &stMessage, NULL,
                      szDate, _countof(szDate));

        GetTimeFormat(LOCALE_USER_DEFAULT, 0,
                      &stMessage, NULL,
                      szTime, _countof(szTime));

        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Timestamp:       %s %s\n"), szDate, szTime);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        SYSTEMTIME stResponse = {0};
        llmClient.ConvertTimestamp(serverResponse.tTimestamp, &stResponse, TRUE);

        GetDateFormat(LOCALE_USER_DEFAULT, 0,
                      &stResponse, NULL,
                      szDate, _countof(szDate));

        GetTimeFormat(LOCALE_USER_DEFAULT, 0,
                      &stResponse, NULL,
                      szTime, _countof(szTime));

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Timestamp:       %s %s\n"), szDate, szTime);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (llmClient.GetNextMessage(&clientMessage, &serverResponse));
}
```

This example uses the **GetDateFormat** and **GetTimeFormat** functions for simplicity and compatibility with both ANSI and Unicode builds. On current Windows platforms, applications should prefer using **GetDateFormatEx** and **GetTimeFormatEx**, which provide improved locale support and are recommended for new development. Note that these newer functions are implemented as Unicode-only APIs.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetFirstMessage, FormatTimestamp, GetNextMessage, SendMessageEx, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::CreateSession Method

```
BOOL CreateSession(
   DWORD dwProviderId,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey,
   LPCTSTR lpszApiVersion
);

BOOL CreateSession(
   DWORD dwProviderId,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey,
   LPCTSTR lpszApiVersion,
   LPCTSTR lpszSystemPrompt,
   LONG nMaxTokens,
   FLOAT fTemperature,
   FLOAT fTopP
);
```

The **CreateSession** method allocates and initializes an internal `LLM_SESSION` structure using the specified connection parameters.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the `LLM_PROVIDER` constants. Refer to the providers page for a list of supported providers and their default models.

lpszBaseUrl

A pointer to a null-terminated string that specifies the base URL for the provider endpoint. This parameter may be NULL to use the default endpoint for the selected provider. Some providers, such as Microsoft Foundry or local services, require a specific base URL.

lpszModelName

A pointer to a null-terminated string that specifies the model name to use for the session. This parameter may be NULL to use the provider default model, if one is defined. Providers without a default model require an explicit model name.

lpszApiKey

A pointer to a null-terminated string that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries.

lpszApiVersion

A pointer to a null-terminated string that specifies the API version required by the provider. This parameter may be NULL if the provider does not require an explicit version. Some providers, such as Anthropic or Microsoft, may require a version string when establishing a connection.

lpszSystemPrompt

A pointer to a null-terminated string that specifies the system prompt used to provide instructions

to the model. This parameter may be NULL if no system prompt is required.

nMaxTokens

The maximum number of tokens that may be generated in the response. Specify zero to use the provider or model default.

fTemperature

The temperature value used to control response variability. Lower values produce more predictable responses, while higher values allow more variation. Specify `LLM_TEMPERATURE_DEFAULT` to use the default value.

fTopP

The nucleus sampling value used to limit token selection based on probability. Specify `LLM_TOP_P_DEFAULT` to use the default behavior. Most applications should leave this value at the default unless they specifically need to tune sampling behavior.

lppSession

A pointer to a variable that receives a pointer to an allocated `LLM_SESSION` structure. On successful return, this parameter will contain a pointer to a structure that can be passed to **ConnectEx**. The structure must be released using the **DeleteSession** method when it is no longer needed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CreateSession** method allocates and initializes an internal `LLM_SESSION` structure for the class instance using the specified parameters. This method is used in conjunction with the **ConnectEx** method when an application requires extended session configuration.

The memory allocated for the session structure will be released when the class destructor is invoked. The **DeleteSession** method can be used to immediately free this memory after a connection has been established, but this is not required.

It is important to note that the class owns the memory allocated for this structure, and there can only be one session structure allocated for each class instance. Your application should never attempt to explicitly free the pointer returned by this method and you should not retain a copy of the pointer. The **GetSession** method can be used to retrieve a pointer to the current session structure if one has been allocated.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    CLlmClient llmClient;
    DWORD dwError = NO_ERROR;

    // Create a session for the class instance
    if (!llmClient.CreateSession(
            LLM_PROVIDER_OPENAI,
            NULL,                       // Default URL
            _T("gpt-5.4-nano"),         // Model name
            _T("%OPENAI_API_KEY%"))     // API key
```

```
    {
        dwError = llmClient.GetLastError();
        _tprintf(_T("Unable to create session, error %lu\n"), dwError);
        return 1;
    }

    // Establish a connection using the ConnectEx method;
    // when called without any arguments, it will use the
    // session that was just created
    if (!llmClient.ConnectEx())
    {
        dwError = llmClient.GetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        return 1;
    }

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse);
    DWORD dwMessageId = 0;

    if (llmClient.SendMessage(
                        _T("What is the capital city of Denmark?"),
                        szResponse,
                        &dwResponseLength,
                        &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    llmClient.Disconnect();
    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectEx, DeleteSession, GetSession, SendMessage, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::DeleteSession Method

```
BOOL DeleteSession();
```

The **DeleteSession** method releases an `LLM_SESSION` structure previously allocated by **CreateSession**.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DeleteSession** method releases the memory allocated for an `LLM_SESSION` structure created by **CreateSession**.

After a session structure has been used with **ConnectEx**, it may be deleted immediately. The library makes an internal copy of the session configuration when the client session is created, so the original structure does not need to remain allocated for the lifetime of the client handle.

This method does not disconnect an active client session. To close a connection created by **Connect** or **ConnectEx**, call **Disconnect**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSession, ConnectEx, Disconnect, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLImClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class methods may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::Disconnect Method

```
INT Disconnect();
```

The **Disconnect** method closes the connection and releases the memory allocated for the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Disconnect** method closes the active connection and releases all memory and resources associated with the specified client session, including the conversation history, cached message and response data, provider information and any other state maintained by the class instance. To preserve information from the session, retrieve it before calling this method.

Avoid calling this method while a blocking request is in progress. If a request must be canceled, call the **Cancel** method first and wait for the blocking operation to return before disconnecting the client session. The **IsBlocking** method can tell you if the class is in the middle of processing a blocking operation, such as retrieving the response to a message prompt.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Cancel, Connect, ConnectEx, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the method and the method's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::EnumModels Method

```
LONG EnumModels(
   LPLLM_MODEL_INFO lpModelList,
   LONG nMaxModels,
   DWORD dwOptions
);
```

The **EnumModels** method returns a list of available models which match the provided options.

Parameters

lpModelList

A pointer to an array of LLM_MODEL_INFO structures that will receive information about the models which match the specified options. This parameter may be NULL, in which case the method only returns the number of matching models.

nMaxModels

The maximum number of LLM_MODEL_INFO structures in the array specified by the ***lpModelList*** parameter. If ***lpModelList*** is NULL, this value should be zero.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
LLM_MODEL_OPTION_DEFAULT	Returns the default list of chat models for the connected provider.
LLM_MODEL_OPTION_PREVIEW	Include preview or experimental models.
LLM_MODEL_OPTION_STREAMING	Include models that support streaming responses.
LLM_MODEL_OPTION_STRUCTURED	Include models that support structured output, such as JSON mode or schema-constrained responses.
LLM_MODEL_OPTION_REASONING	Include models that support reasoning-oriented generation.
LLM_MODEL_OPTION_VISION	Include models that support image input.
LLM_MODEL_OPTION_AUDIO	Include models that support audio output, such as text-to-speech.
LLM_MODEL_OPTION_IMAGE	Include models that support image generation.
LLM_MODEL_OPTION_VIDEO	Include models that support video generation.
LLM_MODEL_OPTION_TRANSCRIPTION	Include models that support audio transcription, such as speech-to-text.
LLM_MODEL_OPTION_DEPRECATED	Include deprecated, retired, or legacy models that are not normally returned.
LLM_MODEL_OPTION_ALL_MODELS	Include all known model types for the connected provider.

Return Value

If the method succeeds, the return value is the number of models which match the specified options. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnumModels** method enumerates the models available for the provider associated with the client session. The client must have been successfully connected before calling this method. The returned list is based on the provider metadata available to the library and, when supported by the provider, information returned by the provider itself.

To determine how many models match the specified options, call this method with the *lpModelList* parameter set to NULL and the *nMaxModels* parameter set to zero. If the method succeeds, the return value is the number of matching models. You can then allocate an array of LLM_MODEL_INFO structures and call the method again to retrieve the model information.

Each LLM_MODEL_INFO structure should have its *dwSize* member initialized to the size of the structure before calling this method. This enables the library to validate the structure version and maintain compatibility with future releases.

The model list may be cached for a limited period of time to avoid unnecessary requests to the provider. Use **GetCacheTime** and **SetCacheTime** to determine or change the cache lifetime for the client session.

The model capabilities reported by this method are intended to help an application select an appropriate model. Providers may change their model availability, capabilities, or status over time, and the provider remains the final authority on whether a specific model can be used for a request.

Example

```
LONG nModels = llmClient.EnumModels(LLM_MODEL_OPTION_DEFAULT);

if (nModels > 0)
{
    LPLLM_MODEL_INFO lpModelList;

    lpModelList = (LPLLM_MODEL_INFO)calloc(nModels, sizeof(LLM_MODEL_INFO));

    if (lpModelList != NULL)
    {
        for (LONG nIndex = 0; nIndex < nModels; nIndex++)
            lpModelList[nIndex].dwSize = sizeof(LLM_MODEL_INFO);

        nModels = llmClient.EnumModels(lpModelList, nModels,
LLM_MODEL_OPTION_DEFAULT);

        if (nModels != LLM_ERROR)
        {
            for (LONG nIndex = 0; nIndex < nModels; nIndex++)
                _tprintf(_T("%s\n"), lpModelList[nIndex].szName);
        }

        free(lpModelList);
    }
}
else if (nModels == LLM_ERROR)
```

```
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to enumerate models, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCacheTime, GetDefaultModelName, GetModelInfo, SetCacheTime, ValidateModel, LLM_MODEL_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::EnumProviders Method

```
LONG EnumProviders(
   LPLLM_PROVIDER_INFO lpProviderList,
   LONG nMaxProviders
);
```

The **EnumProviders** method returns a list of supported service providers.

Parameters

lpProviderList

A pointer to an array of LLM_PROVIDER_INFO structures that will receive information about the supported providers. This parameter may be NULL, in which case the method only returns the number of providers.

nMaxProviders

The maximum number of LLM_PROVIDER_INFO structures in the array specified by the *lpProviderList* parameter. If *lpProviderList* is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of providers. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnumProviders** method returns information about the LLM service providers supported by the library. This includes the provider identifier, provider name, host name, default model name, protocol, port number, and capability flags.

To determine how many providers are available, call this method with the *lpProviderList* parameter set to NULL and the *nMaxProviders* parameter set to zero. If the method succeeds, the return value is the number of providers. You can then allocate an array of LLM_PROVIDER_INFO structures and call the method again to retrieve the provider information.

Each LLM_PROVIDER_INFO structure should have its *dwSize* member initialized to the size of the structure before calling this method. This enables the library to validate the structure version and maintain compatibility with future releases.

The list returned by this method only identifies providers that are known to the library. It does not verify that the provider is currently reachable, that an account is active, or that a valid API key has been configured.

Example

```
LONG nProviders = llmClient.EnumProviders();

if (nProviders > 0)
{
    LPLLM_PROVIDER_INFO lpProviderList;

    lpProviderList = (LPLLM_PROVIDER_INFO)calloc(nProviders,
sizeof(LLM_PROVIDER_INFO));

    if (lpProviderList != NULL)
    {
        for (LONG nIndex = 0; nIndex < nProviders; nIndex++)
```

```
            lpProviderList[nIndex].dwSize = sizeof(LLM_PROVIDER_INFO);

        nProviders = llmClient.EnumProviders(lpProviderList, nProviders);

        if (nProviders != LLM_ERROR)
        {
            for (LONG nIndex = 0; nIndex < nProviders; nIndex++)
                _tprintf(_T("%s\n"), lpProviderList[nIndex].szDisplayName);
        }

        free(lpProviderList);
    }
}
else if (nProviders == LLM_ERROR)
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to enumerate providers, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultModelName, GetProviderFlags, GetProviderInfo, GetProviderName, LLM_PROVIDER_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::EstimateTokenCount Method

```
LONG EstimateTokenCount(
   DWORD dwProviderId
   LPCTSTR lpszMessageText,
   INT cchMessageText
);
```

The **EstimateTokenCount** method estimates the number of tokens in a string.

Parameters

dwProviderId

The identifier of the provider used to estimate the token count. This value determines the tokenization rules that are applied, as different providers may use different tokenization algorithms.

lpszMessageText

A pointer to a null-terminated string containing the text to be analyzed. This parameter cannot be NULL.

cchMessageText

The number of characters in the string specified by the ***lpszMessageText*** parameter. If this value is -1, the string is assumed to be null-terminated.

Return Value

If the method succeeds, the return value is the estimated number of tokens in the specified string. If there are no text characters, or the string only contains whitespace, the method will return zero. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EstimateTokenCount** method provides an approximate count of the number of tokens that would be required to represent the specified text for a given provider. This can be useful when estimating prompt size and ensuring that requests remain within model limits.

The ***lpszMessageText*** parameter must contain valid Unicode text and cannot contain arbitrary binary data or embedded null characters. If malformed Unicode is detected, the method will fail and return LLM_ERROR, with the last error code set to ST_ERROR_INVALID_UNICODE. If the string consists only of whitespace characters, including non-breaking spaces, the method will return zero.

The ANSI version of this method expects the input string to contain either ASCII or UTF-8 encoded text. Strings encoded using a localized ANSI code page are not supported and may fail validation if they contain non-ASCII characters. To ensure internationalized text is processed correctly, it is recommended that your project use Unicode whenever possible.

A token is a unit of text used internally by large language models. Tokens are not the same as characters or words. In general, a token may represent a single word, part of a word, or a short sequence of characters. For example, common words may be represented as a single token, while less common or longer words may be split into multiple tokens. Punctuation and whitespace may also be represented as separate tokens.

As a general rule of thumb, one token corresponds to approximately three to four characters of English text. This can vary significantly depending on the content, language, and tokenization rules used by the provider.

Most providers distinguish between input tokens (the text sent to the model) and output tokens (the

text generated by the model in response). Usage and billing are typically based on the total number of tokens processed, including both the prompt and the generated response. Monitoring token usage can help applications control costs and ensure that requests remain within the limits imposed by the selected model.

The tokenization rules used by each provider can vary, and the exact number of tokens used in a request is determined by the provider at runtime. The value returned by this method is an estimate only and should not be relied on as an exact measurement.

This method does not require an active client session and can be used independently to estimate token usage before establishing a connection or sending a request.

To determine how many input and output tokens have been used in a session, use the **GetTokenUsage** method.

Example

```
LPCTSTR lpszText = _T("Explain how a TCP connection is established.");

LONG nTokens = llmClient.EstimateTokenCount(LLM_PROVIDER_OPENAI, lpszText, -1);

if (nTokens != LLM_ERROR)
{
    _tprintf(_T("Estimated token count: %d\n"), nTokens);
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to estimate token count, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetTokenUsage, NormalizeText, SendMessage, SendMessageEx, ValidateText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ExportHistory Method

```
BOOL ExportHistory(
   LPCTSTR lpszFileName,
   LONG nMessageCount,
   LPLONG lpnExported
);

BOOL ExportHistory(
   LPCTSTR lpszFileName,
   LPLONG lpnExported
);
```

The **ExportHistory** method exports the conversation history to a JSON-formatted file.

Parameters

lpszFileName

A pointer to a null-terminated string which specifies the name of the file to be created. This parameter cannot be NULL or a zero-length string.

nMessageCount

The number of the most recent messages and responses which should be exported. If this parameter is omitted or the value is zero, all messages in the current conversation are exported.

lpnExported

A pointer to an integer which will receive the number of messages which were exported when the method returns. If this information is not required, this parameter may be omitted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ExportHistory** method exports the current conversation history associated with the client session to a JSON-formatted file. The exported history includes both client messages and model responses stored in the current session history.

This method only exports conversation history currently stored in memory for the client session. It cannot be used if the conversation history has been disabled by setting the history size to zero.

If the *lpszFileName* parameter specifies a directory or device name, the method will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters, they will be removed. If the file already exists, the current process must have permission to overwrite the file. If the file has the read-only attribute set, the method will fail and the last error code will be set to `ST_ERROR_ACCESS_DENIED`.

If the *nMessageCount* parameter is non-zero, the method exports only the most recent messages in the conversation history. Older messages are skipped. If the specified count exceeds the number of available messages, all available messages are exported.

The exported file uses UTF-8 encoded JSON text and is intended to be human-readable. Applications should not depend on the exact formatting or internal structure of the file because future versions of the library may include additional metadata or fields. To help prevent data loss, the method writes the

exported history to a temporary file and only replaces the destination file after the export operation has completed successfully. If the export fails, any existing file specified by the *lpszFileName* parameter is left unchanged.

The exported file is intended to provide a simple mechanism for persisting conversation state between application sessions. A previously exported history file can later be restored using the **ImportHistory** method.

Important: The conversation history may contain sensitive or private information. Applications should take reasonable precautions to protect exported history files, such as limiting access permissions or encrypting the exported file. Applications which allow users to export conversation history should consider informing users that the exported file may contain private information.

Example

```
LONG nExported = 0;

// Export the complete conversation history to a JSON file
if (llmClient.ExportHistory(
                    _T("%USERPROFILE%\\Documents\\conversation.json"),
                    &nExported))
{
    _tprintf(_T("Exported %ld messages\n"), nExported);
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Unable to export conversation history, error 0x%08lX\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessageCount, ImportHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::FindMessage Method

```
DWORD FindMessage(
   LPCTSTR lpszPattern,
   DWORD dwStartMessageId,
   DWORD dwOptions
);
```

The **FindMessage** method searches the client conversation history for matching text.

Parameters

lpszPattern

A null-terminated string which specifies the text that should be searched for. This may be a literal string or it may specify a simplified regular expression. The search options can be used to specify if the search is case-sensitive. This parameter cannot be NULL or a string which only contains whitespace.

dwStartMessageId

An unsigned integer which specifies the message ID that the search should begin with. If this parameter value is zero, the search begins with the oldest message in the client history.

dwOptions

An unsigned integer that specifies one or more search options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
LLM_SEARCH_DEFAULT	Search the entire message history for matching text.
LLM_SEARCH_PROMPT	Search the history of message prompts for matching text.
LLM_SEARCH_CASE_SENSITIVE	The pattern match will be case-sensitive. If this option is not specified, matches are not case-sensitive.
LLM_SEARCH_LITERAL_MATCH	Perform a literal string search instead of using a regular expression.

Return Value

If a matching message is found, the method returns a non-zero message ID. If no message prompts and/or responses match the search pattern, the method returns zero. To get extended error information, call **GetLastError**.

Remarks

The **FindMessage** method searches the message history for text that matches a specific pattern. The search can be limited to prompts, responses, or both, depending on the options specified by the *dwOptions* parameter. The pattern you are matching against must be a null-terminated string and it cannot consist of only whitespace characters (spaces, tabs, carriage returns and linefeeds).

If the *dwStartMessageId* parameter is non-zero, the method will begin searching with the first message whose identifier is equal to or greater than the specified value. Message identifiers are unique within a client session and are assigned incrementally, so newer messages will always have higher identifier values than older messages.

If the *lpszPattern* parameter does not contain any regular expression constructs (such as **^**, **$**, **[ ]**, or quantifiers like and), the search is performed using a simple substring comparison. Regular

* +
expression matching is automatically enabled when the pattern contains recognized regular expression constructs, unless the LLM_SEARCH_LITERAL_MATCH option is specified.

If a regular expression pattern is specified, the method compiles the pattern and searches the message content for a match. The regular expression matching uses a simplified version which only supports anchors, wildcards, match classes (enclosed in brackets) and a limited number of escape sequences such as **\d** to match any digit or **\s** to match any space character. It does not support grouping or replacement.

Supported regular expression constructs include:

- The caret **^** and dollar symbol **$** for the beginning and end of the content.
- The period **.** to match any character.
- Character classes such as **[abc]** which would match any of the characters enclosed in brackets.
- Negated classes such as **[^abc]** which would match if none of those characters are found.
- Escaped matches such as **\d** for digits, **\w** for word characters and **\s** for whitespace characters
- The **+** quantifier can be used to match one or more occurrences.
- The ***** quantifier will match zero or more occurrences.

This method uses a minimal regular expression parser and does not include support for complex constructs such as grouping, backreferences or alternation using the pipe **|** character. If you want to search for text which may contain characters typically used in regular expressions such as **[** or **$**, you can either escape them using a backslash, or you can use the LLM_SEARCH_LITERAL_MATCH option which forces the method to only perform a substring search.

Applications searching for arbitrary user-entered text should usually specify the LLM_SEARCH_LITERAL_MATCH option to avoid unintended interpretation of regular expression characters. This will avoid situations where a user may unexpectedly enter search text which appears to be a regular expression pattern.

Regular expression matches are performed against the entire prompt or response text and are not implicitly anchored. It is also important to keep in mind that matches can span multiple lines (they do not end at line breaks) and will attempt to match as much text as possible (this is sometimes referred to as a "greedy match"). This can yield unexpected results if you use wildcards in the pattern which are overly broad, matching large sections of the prompt or response text. For example, a pattern like ".*abc" will match everything until the last instance of "abc" is found.

If you are calling this method in C, C++ or another C-like language and want to use a backslash in your regular expression, remember you will need to escape the backslash (in other words, use double backslashes). For example, if you wanted to use the regular expression \d+\s+ in your pattern, the C string literal would look like "\\d+\\s+", otherwise the compiler will probably warn you about unrecognized escape sequences.

If neither the LLM_SEARCH_PROMPT nor LLM_SEARCH_RESPONSE options are explicitly specified, the search will match against both message prompts and responses.

The **GetMessageById** method can be used to obtain the complete contents of the prompt and/or response for a matching message.

Example

```
LPCTSTR lpszPattern = _T("exception.*handler");
DWORD dwOptions = LLM_SEARCH_DEFAULT;
DWORD dwMessageId = 0;
```

```
// Find the first message that contains the specified text
dwMessageId = llmClient.FindMessage(lpszPattern, dwMessageId, dwOptions))

if (dwMessageId != 0)
{
    _tprintf(_T("Found match in message ID %lu\n"), dwMessageId);

    // Continue searching for additional matches
    do
    {
        dwMessageId = llmClient.FindMessage(lpszPattern, dwMessageId + 1,
dwOptions);

        if (dwMessageId != 0)
            _tprintf(_T("Found another match in message ID %lu\n"), dwMessageId);
    }
    while (dwMessageId != 0);
}
else
{
    _tprintf(_T("No matching messages were found.\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstMessage, GetHistorySize, GetMessageById, GetMessageCount, GetNextMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::FormatTimestamp Method

```
BOOL INT LlmFormatTimestamp(
   TIME_T tTimestamp,
   LPTSTR lpszDateTime,
   INT nMaxLength,
   BOOL bLocalTime
);

BOOL INT LlmFormatTimestamp(
   TIME_T tTimestamp,
   CString &strDateTime,
   BOOL bLocalTime
);
```

The **FormatTimestamp** method returns a timestamp value as an ISO 8601 formatted date and time string.

Parameters

tTimestamp

A 64-bit timestamp value associated with a message or response. This value represents the number of seconds that have elapsed since January 1, 1970 UTC.

lpszDateTime

A pointer to a buffer that receives the formatted date and time string. This parameter cannot be NULL and must be large enough to store the entire string, including the terminating null character. An alternate version of this method accepts a **CString** object which will contain the formatted timestamp when the method returns.

nMaxLength

The size of the buffer specified by the lpszDateTime parameter, in characters. This value must be large enough to store the formatted string. A buffer of at least 32 characters is recommended.

bLocalTime

If this parameter is non-zero, the timestamp is converted to the local system time before formatting. If this parameter is zero, the timestamp is formatted as Coordinated Universal Time (UTC).

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **FormatTimestamp** method converts a timestamp value returned by the library into a human-readable date and time string using the ISO 8601 format. This format is consistent, unambiguous, and suitable for logging, display, or storage.

The *tTimestamp* parameter is a standard 64-bit Unix time value and can be used interchangeably with functions in the C standard library such as `gmtime` or `localtime`. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps are returned by methods such as **SendMessage** and can be converted into a local date and time using the **ConvertTimestamp** method.

The formatted string is returned in the form "YYYY-MM-DDThh:mm:ssZ" when UTC is specified. When local time is specified, the string is returned in the form "YYYY-MM-DDThh:mm:ss+hh:mm" or "YYYY-MM-DDThh:mm:ss-hh:mm", depending on the current time zone offset.

When ***bLocalTime*** is non-zero, the formatted string includes the local time zone offset. This ensures that the resulting value is not ambiguous and preserves the ISO 8601 format.

Applications that require locale-specific formatting or alternative date and time representations should convert the timestamp using **ConvertTimestamp** and apply additional formatting as needed. If you need to display the date and time using conventions for the current locale, use functions such as **GetDateFormatEx** and **GetTimeFormatEx** in the Windows API.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };

if (llmClient.GetFirstMessage(&clientMessage, &serverResponse))
{
    do
    {
        TCHAR szDateTime[64] = {0};

        llmClient.FormatTimestamp(clientMessage.tTimestamp, szDateTime,
_countof(szDateTime), FALSE);

        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Timestamp:       %s\n"), szDateTime);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        llmClient.FormatTimestamp(serverResponse.tTimestamp, szDateTime,
_countof(szDateTime), FALSE);

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Timestamp:       %s\n"), szDateTime);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (llmClient.GetNextMessage(&clientMessage, &serverResponse));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTimestamp, GetFirstMessage, GetNextMessage, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetCacheTime Method

```
INT GetCacheTime();
```

The **GetCacheTime** method returns the current cache lifetime, in seconds, for the specified client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the current cache time in seconds. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetCacheTime** method retrieves the amount of time, in seconds, that provider and model information is cached by the client session. Caching this information reduces the number of requests made to the provider and can improve performance when repeatedly querying for models or provider metadata.

By default, the cache lifetime for a client session is one hour (3600 seconds). The cache time can be adjusted using the **SetCacheTime** method. Shorter cache times will refresh provider and model information more frequently, while longer cache times can reduce network activity.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetIdleTime GetProviderInfo, GetTimeout, SetCacheTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetDefaultModelFlags Method

```
BOOL GetDefaultModelFlags(
   DWORD dwProviderId
   LPCTSTR lpszModelName,
   LPDWORD lpdwFlags
);
```

The **GetDefaultModelFlags** method returns the default capability flags for the specified model.

Parameters

dwProviderId

The identifier of the provider associated with the model.

lpszModelName

A null-terminated string which contains the name of the new model. If this parameter is NULL or a zero-length string, the default model for the specified provider will be used.

lpdwFlags

A pointer to an unsigned integer which may have one or more bit flags set when the function returns. If there are no capabilities defined for the specified model, this value will be set to `LLM_MODEL_FLAG_NONE`. This parameter cannot be NULL and the value does not need to be initialized by the caller.

Constant	Description
LLM_MODEL_FLAG_NONE	No model capabilities defined
LLM_MODEL_FLAG_CHAT	Supports conversational text generation
LLM_MODEL_FLAG_SYSTEM_ROLE	Accepts a system role message in the input
LLM_MODEL_FLAG_STREAMING	Supports streaming responses
LLM_MODEL_FLAG_STRUCTURED	Supports structured output (e.g., JSON or schema)
LLM_MODEL_FLAG_VISION	Supports image inputs (multimodal)
LLM_MODEL_FLAG_TOOLS	Supports tool or function calling
LLM_MODEL_FLAG_REASONING	Supports reasoning-oriented generation
LLM_MODEL_FLAG_RESPONSES	Supports the Responses API (unified endpoint)
LLM_MODEL_FLAG_FUNCTIONS	Supports OpenAI-style function calling with schemas
LLM_MODEL_FLAG_AUDIO	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	Supports image generation
LLM_MODEL_FLAG_VIDEO	Supports video generation
LLM_MODEL_FLAG_TRANSCRIPTION	Supports speech-to-text transcription

LLM_MODEL_FLAG_INSTRUCTIONS	Supports a dedicated instruction field outside message lists
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	Uses `max_completion_tokens` instead of `max_tokens` in requests
LLM_MODEL_FLAG_INFERRED	Capabilities are inferred rather than explicitly defined

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **GetDefaultModelFlags** method returns the default capability flags for a model associated with the specified provider. If the model name is recognized, the library will return the inferred capabilities associated with that model. If the model name is not recognized, the *dwFlags* parameter will return `LLM_MODEL_FLAG_NONE`.

This method does not require the application to be connected to a service provider. These capabilities are inferred from an internal table of recognized models and may not always reflect the current behavior of a provider, particularly when providers update, rename or replace models. The **GetModelFlags** method can be used to determine the capabilities of the model that was selected when a connection was established with a provider.

Because the capability flags are derived from the model name rather than explicitly provided by the service provider, the `LLM_MODEL_FLAG_INFERRED` flag will be set for recognized models.

To obtain the name of the default model the library uses for a specific provider, call the **GetDefaultModelName** method.

Example

```
DWORD dwProviderId = LLM_PROVIDER_OPENAI;
TCHAR szModelName[LLM_MODEL_NAMELEN] = _T("");

if (llmClient.GetDefaultModelName(dwProviderId, szModelName, _countof(szModelName))
> 0)
{
    DWORD dwFlags = LLM_MODEL_FLAG_NONE;

    if llmClient.GetDefaultModelFlags(dwProviderId, szModelName, &dwFlags))
    {
        if (dwFlags & LLM_MODEL_FLAG_REASONING)
            _tprintf(_T("The %s model has reasoning capability\n"), szModelName);
    }
    else
    {
        DWORD dwError = llmClient.GetLastError();
       _tprintf(_T("Unable to retrieve default model flags, error 0x%08lx\n"),
dwError);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumModels, GetDefaultModelName, GetModelInfo, ValidateModel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetDefaultModelName Method

```
INT GetDefaultModelName(
   DWORD dwProviderId
   LPCTSTR lpszModelName,
   INT nMaxLength
);

INT GetDefaultModelName(
   DWORD dwProviderId
   CString &strModelName
);
```

The **GetDefaultModelName** method returns the name of the default model for a service provider.

Parameters

dwProviderId

The identifier of the provider whose default model name will be returned.

lpszModelName

A pointer to a buffer that receives the default model name. This parameter may be NULL, in which case the method returns the length of the model name without copying it. An alternate version of this method accepts a **CString** object which will contain the model name when the method returns.

nMaxLength

The size of the buffer specified by the ***lpszModelName*** parameter, in characters. If ***lpszModelName*** is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDefaultModelName** method returns the default model name associated with the specified provider. This is the model that will be used if a session is created or connected without explicitly specifying a model name. In most cases, the default model is selected to provide a balance of performance and cost, typically favoring smaller, faster models that are suitable for general use. For more information about providers and their default models, refer to the Large Language Model Providers section.

Applications can call this method to determine a suitable default model for a provider, or to present a default selection to the user. The default model may change over time as providers update or replace their models.

If the ***lpszModelName*** parameter is NULL, the method returns the length of the model name, in characters, not including the terminating null character. This allows an application to determine the required buffer size before retrieving the string.

The model name is returned as a null-terminated string. If a buffer is provided but is not large enough to receive the entire string, the method fails and sets the error code to ST_ERROR_BUFFER_TOO_SMALL. A buffer of 128 characters is typically sufficient for most model names.

Example

```
INT cchModelName = llmClient.GetDefaultModelName(LLM_PROVIDER_OPENAI);

if (cchModelName > 0)
{
    LPTSTR lpszModelName;
    lpszModelName = (LPTSTR)malloc((cchModelName + 1) * sizeof(TCHAR));

    if (lpszModelName != NULL)
    {
        if (llmClient.GetDefaultModelName(LLM_PROVIDER_OPENAI, lpszModelName,
cchModelName + 1) > 0)
        {
             _tprintf(_T("The default model is %s\n"), lpszModelName);
        }

        free(lpszModelName);
    }
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Unable to retrieve default model name, error 0x%08lx\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumModels, GetModelInfo, ValidateModel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetDefaultUrl Method

```
INT GetDefaultUrl(
   DWORD dwProviderId,
   LPCTSTR lpszBaseUrl,
   INT nMaxLength
);

INT GetDefaultUrl(
   DWORD dwProviderId,
   CString &strBaseUrl
);
```

The **GetDefaultUrl** method returns the default endpoint URL for the specified provider.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpszBaseUrl

A pointer to a buffer that receives the default base endpoint URL used when making requests to the provider. If this parameter is NULL, the method will return the length of the URL without copying to the string buffer. An alternate version of this method accepts a reference to a **CString** object which will contain the endpoint URL when the method returns.

nMaxLength

The size of the buffer specified by the ***lpszBaseUrl*** parameter, in characters. If ***lpszBaseUrl*** is NULL, this value should be zero. If a buffer is provided, it must be large enough to store the complete URL, including the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDefaultUrl** method returns the default base endpoint URL used when a client makes a request to the provider. If the provider ID is unknown or does not have a default endpoint, this method will return zero. If the provider does not have a default endpoint, the last error code will be set to ST_ERROR_NO_PROVIDER_BASE_URL. In this case, a client must always provide a URL endpoint.

To return the base endpoint URL being used with the current client session, use the **GetProviderUrl** method. The **GetDefaultUrl** method does not require an active client session and will only return the default endpoint for a specific provider. It will not return a custom endpoint specified for a provider.

The base URL is returned as a null-terminated string. If the buffer is not large enough to receive the entire string, the method fails and sets the error code to ST_ERROR_BUFFER_TOO_SMALL. We recommend a buffer size of at least 256 characters when retrieving the base URL.

Example

```
DWORD dwProviderId = LLM_PROVIDER_OPENAI;
TCHAR szDefaultUrl[256] = {0};
```

```
INT cchDefaultUrl = llmClient.GetDefaultUrl(dwProviderId, szDefaultUrl,
_countof(szDefaultUrl));

if (cchDefaultUrl > 0)
{
    _tprintf(_T("The default endpoint for provider %lu is %s\n"), dwProviderId,
szDefaultUrl);
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to get the provider endpoint, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultModelName, GetProviderFlags, GetProviderId, GetProviderInfo, GetProviderName, GetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString &strDescription
);
```

The **GetErrorString** method returns the description of a specific error code.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Remarks

This method is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages. Applications should not rely on the returned description text for program logic. Use the numeric error code for comparisons and handling specific conditions.

The error descriptions returned by this method are provided in English only. Applications that require localized error messages should provide their own translations based on the returned error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetFirstMessage Method

```
BOOL GetFirstMessage(
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **GetFirstMessage** method returns the first available message in the conversation history.

Parameters

lpMessage

A pointer to an LLM_MESSAGE structure that receives information about the client request. This parameter may be NULL, in which case no message information is returned.

lpResponse

A pointer to an LLM_RESPONSE structure that receives information about the server response. This parameter may be NULL, in which case no response information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**. If there are no messages in the client history, the error code will be set to `ST_ERROR_END_OF_MESSAGES`.

Remarks

The **GetFirstMessage** method retrieves the first message and associated response from the client conversation history. This method is typically used in conjunction with the **GetNextMessage** method to enumerate all messages in the session.

If the *lpMessage* and/or *lpResponse* parameters are specified, their respective structures must be initialized with the *dwSize* member set to the size of the structure before calling this method. This enables the library to validate the structure version and maintain compatibility with future releases.

Messages are returned in chronological order, beginning with the oldest message in the client history. New messages are always appended to the end of the history, and if the history is trimmed, older messages are removed first. The order of messages is stable and reflects the sequence in which they were submitted.

If there are no messages in the client history, the method fails and sets the error code to ST_ERROR_END_OF_MESSAGES. Applications can use this condition to determine that no messages are available for enumeration.

Either the *lpMessage* or *lpResponse* parameter may be NULL if only one type of information is required. At least one of these parameters must be specified. If both parameters are NULL, the method fails and sets the error code to ST_ERROR_INVALID_PARAMETER.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };

if (llmClient.GetFirstMessage(&clientMessage, &serverResponse))
{
    do
    {
```

```
        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (llmClient.GetNextMessage(&clientMessage, &serverResponse));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetNextMessage, GetHistorySize, GetMessageById

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetHeader Method

```
INT GetHeader(
   LPCTSTR lpszHeaderName,
   LPTSTR lpszHeaderValue,
   INT nMaxLength
);
```

The **GetHeader** method returns the value of an HTTP header from the server response.

Parameters

lpszHeaderName

A pointer to a null-terminated string which specifies the name of the HTTP response header. The header names are not case-sensitive. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszHeaderValue

A pointer to a string buffer which will contain the header value when the method returns. If this parameter is NULL, the header value will not be copied and the method will return the length. If the buffer is not large enough to contain the entire header value, it will be truncated.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderValue* string buffer. If *lpszHeaderValue* is NULL, this value should be zero. The maximum length must include room for the terminating null character.

Return Value

If the method succeeds, the return value is the length of the header value in characters, not including the terminating null character. If the header is undefined or the method fails, the return value is zero. To get extended error information, call **GetLastError**. If the header field does not exist, the error code will be `ST_ERROR_HEADER_NOT_FOUND`.

Remarks

The **GetHeader** method returns the value of a response header from the last request made to the service provider.

This method is intended for applications that need lower-level access to provider-specific response headers. Most applications do not need to call this method. Standard response information, such as the HTTP status code, response text, token usage, model name, and elapsed time, is available through the API.

The headers returned by a service provider can vary depending on the provider, model, endpoint, account configuration, and the specific request being made. Applications should not assume that a particular header will always be present unless it is documented by the service provider.

If the header field specified by *lpszHeaderName* was not returned by the server, this method will return zero. If the *lpszHeaderValue* parameter is not NULL, it will contain an empty string when the method returns. The response headers are only available after a request has been sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ResetSession, SendMessage, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetHistorySize Method

```
LONG GetHistorySize();
```

The **GetHistorySize** method returns the maximum number of message exchanges that can be stored in the conversation history.

Parameters

None.

Return Value

If the method succeeds, the return value is the maximum size of the conversation history for the client session. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetHistorySize** method returns the maximum number of messages which can be stored in the client conversation history. Each message represents a single request submitted to the model and its associated response.

The conversation history is maintained on a per-session basis and reflects the sequence of interactions performed using the client handle. Messages are stored in chronological order, with new messages appended to the end of the history. To retrieve the contents of a specific message and response from the history, use either the **GetMessage** or **GetMessageById** methods.

It is possible to disable the conversation history by calling the **SetHistorySize** method and specifying a maximum history size of zero. In this case, this method will always return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetFirstMessage, GetMessage, GetMessageById, GetNextMessage, SetHistorySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetIdleTime Method

```
INT GetIdleTime();
```

The **GetIdleTime** method returns the current idle time for the specified client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the current idle timeout period in seconds. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetIdleTime** method returns the current idle timeout period, in seconds, for the specified client session. This value determines how long a connection can remain idle between requests before the library automatically closes and re-establishes the underlying HTTP connection.

Idle time is used to balance performance and reliability. Reusing an existing connection can reduce overhead and improve performance, particularly when multiple requests are made in quick succession. However, connections that remain idle for extended periods may be closed by the server or by intermediate network infrastructure. When the idle time is exceeded, the library will proactively establish a new connection before sending the next request.

This value is distinct from the timeout period and cache time. The timeout period specifies how long the client will wait for a response to a request before failing, while the cache time controls how long certain responses or metadata may be reused without making a new request. The idle time only affects how long an existing connection is retained between requests.

The idle time can be modified using the **SetIdleTime** method. If a value of zero is specified, a new connection is established for each request. In most cases, the default idle time provides a good balance between performance and compatibility.

It is important to note that this setting only affects client-side behavior. Increasing the idle time does not guarantee that a server will keep the connection open for that duration. Many servers will close idle connections after a shorter period of time, in which case the library will automatically reconnect as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetCacheTime, GetTimeout, SetIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetLastError Method

```
DWORD GetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call **GetLastError** immediately when a method's return value indicates that an error has occurred. Functions set this value by calling **SetLastError**. The Return Value section of each reference page notes the conditions under which a function sets the last error code.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or LLM_ERROR. Those functions which call **SetLastError** when they succeed are noted on the method reference page.

This method should only be called when a preceding function indicates failure. Calling this method after a successful call may return a value unrelated to the most recent operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetErrorString, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetMemoryUsage Method

```
BOOL GetMemoryUsage(
   PLARGE_INTEGER lpMemUsage
);

BOOL GetMemoryUsage(
   LONGLONG *lpMemUsage
);
```

The **GetMemoryUsage** method returns the number of bytes of memory allocated for the client session.

Parameters

lpMemUsage

A pointer to a **LARGE_INTEGER** variable which will contain the number of bytes allocated for the client session when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetMemoryUsage** method returns an approximate amount of memory currently allocated for the specified client session. This includes the client session structure, dynamically allocated session data, cached model information, and message history retained by the session.

This method is primarily intended for diagnostics, monitoring, and testing. Applications that maintain long-running sessions or retain large message histories can use this method to monitor how much memory is being used by a client session.

The value returned by this method is rounded up to the nearest system page size and should be treated as an estimate of memory usage rather than an exact byte-for-byte allocation total.

The *lpMemUsage* parameter points to a LARGE_INTEGER structure. The memory usage value is returned in the QuadPart member. In languages that support 64-bit integer values directly, it is safe to pass a pointer to a compatible 64-bit integer variable.

The memory usage reported by this method is limited to memory allocated for the specified client session. It does not include memory used internally by the operating system, networking libraries, provider services, or other process-wide resources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

ClearHistory, GetHistorySize, SetHistorySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetMessage Method

```
BOOL LlmGetMessage(
   LONG nMessageIndex,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **GetMessage** method retrieves a message turn in the client history using an index value.

Parameters

nMessageIndex

An integer value which identifies the message using a zero-based index value. This value cannot be less than zero or greater than the number of messages in the conversation history.

lpMessage

A pointer to an LLM_MESSAGE structure that receives information about the client request. This parameter may be NULL, in which case no message information is returned.

lpResponse

A pointer to an LLM_RESPONSE structure that receives information about the server response. This parameter may be NULL, in which case no response information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetMessage** method retrieves a message and its corresponding response from the client history using a zero-based index value. The index value of zero refers to the oldest message currently stored in the history.

The message history is managed as a first-in, first-out (FIFO) queue. When the number of messages exceeds the configured history size, older messages are automatically removed. As a result, the index value for a given message may change over time. Applications that need to reliably reference a specific message should use the **GetMessageById** method instead.

If message history has been disabled, the library does not retain previous messages. However, the most recent message and response are still available and may be retrieved using an index value of zero. This also means the data returned in the LLM_MESSAGE and LLM_RESPONSE structures will only remain valid until the next call to **SendMessage**, **SendMessageEx**, or until the session is reset or disconnected. Applications that need to preserve this information should make a copy of the data before issuing another request.

If the *lpMessage* and/or *lpResponse* parameters are specified, their respective structures must be initialized with the *dwSize* member set to the size of the structure before calling this method. This enables the library to validate the structure version and maintain compatibility with future releases.

Either the *lpMessage* or *lpResponse* parameter may be NULL if only one type of information is required. At least one of these parameters must be specified. If both parameters are NULL, the method fails and sets the error code to ST_ERROR_INVALID_PARAMETER.

Example

```
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE) };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE) };
LONG nMessageIndex = 0;

// If conversation history is enabled, retrieve the most recent message.
// If history is disabled, an index of 0 refers to the most recent
// message and response from the previous request.

LONG nCount = llmClient.GetMessageCount();

if (llmClient.GetHistorySize() > 0 && nCount > 0)
    nMessageIndex = nCount - 1;

if (llmClient.GetMessage(nMessageIndex, &clientMessage, &serverResponse))
{
    _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
    _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
    _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
    _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

    _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
    _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
    _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
    _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
    _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
    _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
    _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
    _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Unable to retrieve message at index %ld (error 0x%08lX)\n"),
nMessageIndex, dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstMessage, GetMessageById, GetNextMessage, SendMessage, SendMessageEx, SetHistorySize, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetMessageById Method

```
BOOL LlmGetMessageById(
   DWORD dwMessageId,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **GetMessageById** method retrieves a message turn in the client history based on a numeric identifier.

Parameters

dwMessageId

A numeric value which identifies a specific message and response.

lpMessage

A pointer to an LLM_MESSAGE structure that receives information about the client request. This parameter may be NULL, in which case no message information is returned.

lpResponse

A pointer to an LLM_RESPONSE structure that receives information about the server response. This parameter may be NULL, in which case no response information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageById** method retrieves a specific message and its associated response from the client session history. This method can be used when an application has stored a message identifier or has obtained one using methods such as **FindMessage**. Message identifiers are returned when sending requests using **SendMessage** or **SendMessageEx**.

If message history has been disabled, the library does not retain previous messages. However, the most recent message and response are still available and may be retrieved specifying its message identifier. This also means the data returned in the LLM_MESSAGE and LLM_RESPONSE structures will only be valid until the next call to **SendMessage**, **SendMessageEx**, or until the session is reset or disconnected. Applications that need to preserve this information should make a copy of the data before issuing another request.

If the *lpMessage* and/or *lpResponse* parameters are specified, their respective structures must be initialized with the *dwSize* member set to the size of the structure before calling this method. This enables the library to validate the structure version and maintain compatibility with future releases.

Either the *lpMessage* or *lpResponse* parameter may be NULL if only one type of information is required. At least one of these parameters must be specified. If both parameters are NULL, the method fails and sets the error code to ST_ERROR_INVALID_PARAMETER.

Message identifiers are session-specific and are guaranteed to be unique only within the client session in which they were created. Applications must not rely on the internal assignment strategy used to generate these identifiers. The only guaranteed behavior is that message identifiers are assigned values which ensure newer messages always receive larger values than older messages.

If the specified message identifier does not match a message currently stored in the client history, the

method fails. This can occur if the message has been removed because the history was cleared or trimmed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindMessage, GetFirstMessage, GetMessage, GetNextMessage, SendMessage, SendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetMessageCount Method

```
LONG GetMessageCount();
```

The **GetMessageCount** method returns the current number of messages in the client history.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of messages in the client history. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageCount** method returns the number of messages currently stored in the client session history. Each message represents a single request submitted to the model and its associated response.

The conversation history is maintained on a per-session basis and reflects the sequence of interactions performed using the client handle. Messages are stored in chronological order, with new messages appended to the end of the history. To retrieve the contents of a specific message and response from the history, use either the **GetMessage** or **GetMessageById** methods.

It is possible to disable the conversation history by calling the **SetHistorySize** method and specifying a maximum history size of zero. In this case, this method will always return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

ClearHistory, GetMessage, GetMessageById, SetHistorySize, TrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetModelContextSize Method

```
LONG GetModelContextSize(
   LPCTSTR lpszModelName
);
```

The **GetModelContextSize** method returns the context size for the specified model.

Parameters

lpszModelName

A pointer to a null-terminated string that specifies the name of the model. This parameter may be NULL or an empty string, in which case the context size is returned for the model selected for the current client session. If a model name is specified, it must be valid for the provider associated with the client session.

Return Value

If the method succeeds, the return value is the context size for the model. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetModelContextSize** method returns the maximum number of tokens that can be processed by the specified model in a single request. This value is commonly referred to as the model's context size or context window.

The context size includes both the input provided to the model and the output generated in response. This means that the total number of tokens in a request, including the prompt, conversation history, and generated response, must not exceed this limit.

If the combined number of tokens exceeds the model's context size, the request may be rejected by the provider or the response may be truncated. Applications should take this limit into account when constructing prompts or maintaining long conversation histories.

In general, larger context sizes allow a model to retain more information from previous interactions, but may also increase latency and cost depending on the provider and model being used.

The context size returned by this method should be treated as an approximate limit rather than an absolute value. For some providers, particularly locally hosted models, the effective context size may be configured or adjusted outside of the application. In these cases, the reported value serves as a guideline and may not reflect the exact limit enforced by the model at runtime.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EstimateTokenCount, GetModelInfo, GetTokenUsage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetModelFlags Method

```
BOOL GetModelFlags(
   LPDWORD lpdwFlags
);
```

The **GetModelFlags** method returns the flags associated with the current model.

Parameters

lpdwFlags

A pointer to an unsigned integer which may have one or more bit flags set when the method returns. If there are no capabilities defined for the provider, this value will be set to **LLM_MODEL_FLAG_NONE**. This parameter cannot be NULL and the value does not need to be initialized by the caller.

Name	Description
LLM_MODEL_FLAG_NONE	No model capabilities defined
LLM_MODEL_FLAG_CHAT	Supports conversational text generation
LLM_MODEL_FLAG_SYSTEM_ROLE	Accepts a system role message in the input
LLM_MODEL_FLAG_STREAMING	Supports streaming responses
LLM_MODEL_FLAG_STRUCTURED	Supports structured output (e.g., JSON or schema)
LLM_MODEL_FLAG_VISION	Supports image inputs (multimodal)
LLM_MODEL_FLAG_TOOLS	Supports tool or function calling
LLM_MODEL_FLAG_REASONING	Supports reasoning-oriented generation
LLM_MODEL_FLAG_RESPONSES	Supports the Responses API (unified endpoint)
LLM_MODEL_FLAG_FUNCTIONS	Supports OpenAI-style function calling with schemas
LLM_MODEL_FLAG_AUDIO	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	Supports image generation
LLM_MODEL_FLAG_VIDEO	Supports video generation
LLM_MODEL_FLAG_TRANSCRIPTION	Supports speech-to-text transcription
LLM_MODEL_FLAG_INSTRUCTIONS	Supports a dedicated instruction field outside message lists
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	Uses **max_completion_tokens** instead of **max_tokens** in requests
LLM_MODEL_FLAG_INFERRED	Capabilities are inferred rather than explicitly defined

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetModelFlags** method returns a set of bit flags that describe the capabilities supported by the current model associated with the client session. These flags can be used by applications to determine which features are available and adjust behavior accordingly.

Model capabilities may vary between providers and even between models offered by the same provider. Not all providers return complete capability information when enumerating models. In these cases, some flags may be inferred by the library based on the model name and known characteristics of the provider.

Because capability information may be incomplete or inferred, applications should treat these flags as guidance rather than absolute guarantees of behavior. The provider ultimately determines which features are supported and how they are implemented.

The value returned in the ***lpdwFlags*** parameter is a bitmask and may contain multiple flags combined using a bitwise OR operation. If no capabilities are defined or available for the model, the value will be set to `LLM_MODEL_FLAG_NONE`. In practice, most supported models provide at least basic conversational capabilities, so a value of zero is uncommon and typically indicates that capability information is not available or could not be determined.

In some cases, capability information may be incomplete or inferred. If necessary, applications can override the detected capabilities using the **SetModelFlags** method. This can be useful when working with providers or models that require specific options which are not automatically detected. In general, applications should rely on the default capability detection unless a specific compatibility issue is encountered.

Some flags may be reserved for internal use or represent provider-specific capabilities. Applications should not rely on undocumented flag values and should only test for defined LLM_MODEL_FLAG constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

EnumModels, GetModelContextSize, GetModelInfo, GetModelName, SetModelFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetModelInfo Method

```
BOOL GetModelInfo(
   LPCTSTR lpszModelName,
   LPLLM_MODEL_INFO lpModelInfo
);
```

The **GetModelInfo** method returns information about a specific model for the current provider.

Parameters

lpszModelName

A pointer to a null-terminated string that specifies the name of the model. This parameter may be NULL or an empty string, in which case information about the current model associated with the client session is returned.

lpModelInfo

A pointer to an LLM_MODEL_INFO structure that receives information about the model. This parameter cannot be NULL. The ***dwSize*** member of the structure must be initialized before calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetModelInfo** method retrieves detailed information about a model associated with the provider used by the client session. If a model name is specified, it must be valid for the current provider. If no model name is specified, the method returns information about the model currently associated with the session.

The ***lpModelInfo*** parameter must point to an LLM_MODEL_INFO structure whose ***dwSize*** member has been initialized to `sizeof(LLM_MODEL_INFO)`. This enables the library to validate the structure version and maintain compatibility with future releases.

The information returned by this method includes the model identifier, provider, capability flags, status flags, token limits, context size, and optional metadata such as display name, owner, version, and creation date. Not all providers supply complete metadata for every model, and some values may be empty or unspecified.

Model capability and limit information should be treated as descriptive metadata rather than an absolute guarantee of behavior. Providers may change model availability, limits, or capabilities over time, and locally hosted models may allow some values, such as context size, to be configured outside of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumModels, GetModelFlags, GetModelContextSize, LLM_MODEL_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetNextMessage Method

```
BOOL GetNextMessage(
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **GetNextMessage** method returns the next available message in the client history.

Parameters

lpMessage

A pointer to an LLM_MESSAGE structure which receives information about the next message in the client history. This parameter may be NULL if the message information is not needed.

lpResponse

A pointer to an LLM_RESPONSE structure which receives information about the response associated with the message. This parameter may be NULL if the response information is not needed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**. When the last message in the history is reached, the error code will be set to `ST_ERROR_END_OF_MESSAGES`.

Remarks

The **GetNextMessage** method is used to enumerate the messages stored in the client history. Each successful call advances the internal enumeration position and returns information about the next message and, if available, the response associated with that message.

If the *lpMessage* or *lpResponse* parameters are specified, the structures must be initialized before calling this method. The dwSize member must be set to the size of the corresponding structure. This enables the method to validate the structure version and safely return the requested information.

Either *lpMessage* or *lpResponse* may be NULL. This is useful when an application only needs to enumerate the original messages or only needs information about the responses. At least one of these parameters should be specified.

To begin enumerating messages, call **GetFirstMessage** to retrieve the first message in the client history. Subsequent messages can then be retrieved by calling **GetNextMessage** until the method returns zero and the last error code is set to ST_ERROR_END_OF_MESSAGES. This condition indicates the enumeration has completed and should not be treated as a failure.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };

if (llmClient.GetFirstMessage(&clientMessage, &serverResponse))
{
    do
    {
        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);
```

```
        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (llmClient.GetNextMessage(&clientMessage, &serverResponse));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstMessage, GetMessageCount, SendMessage, TrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetOptions Method

```
BOOL GetOptions(
   LPDWORD lpdwOptions
);
```

The **GetOptions** method returns the current client option flags.

Parameters

lpdwOptions

A pointer to an unsigned integer that will be set to the current client options when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetOptions** method returns the current option flags for the client session. The variable specified by the *lpdwOptions* parameter will be initialized to LLM_OPTION_NONE and then updated with the client options before the method returns. The value may contain one or more of the following flags combined with a bitwise OR operation.

Value	Description
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be used when a provider does not require a User-Agent header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base

	URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.

This method can be used in conjunction with **SetOptions** to query and modify client behavior at runtime.

Example

```
DWORD dwProviderId = LLM_PROVIDER_NONE;
TCHAR szBaseUrl[MAX_PATH] = {0};

// Determine the provider ID for the current session and get
// the base URL used for API requests
dwProviderId = llmClient.GetProviderId();

if (dwProviderId == LLM_PROVIDER_NONE)
{
    _tprintf(_T("Unable to get provider ID for this session\n"));
    return;
}

if (llmClient.GetProviderUrl(szBaseUrl, _countof(szBaseUrl)) < 1)
{
    _tprintf(_T("Unable to get provider endpoint for this session\n"));
    return;
}

// Get the current client options
DWORD dwOptions = LLM_OPTION_NONE;

if (!llmCient.GetOptions(&dwOptions))
{
    _tprintf(_T("Unable to get client options for this session\n"));
    return;
}

// Get information about the current provider
LLM_PROVIDER_INFO providerInfo = { sizeof(providerInfo), 0 };

if (!llmClient.GetProviderInfo(dwProviderId, &providerInfo))
{
    _tprintf(_T("Unable to get provider information\n"));
    return;
}

_tprintf(_T("Connected to %s using %s, the connection is %s\n"),
         providerInfo.szDisplayName,
         szBaseUrl,
         ((dwOptions & LLM_OPTION_SECURE) ? _T("secure") : _T("not secure")));
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetProviderId, GetProviderInfo, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetProviderFlags Method

```
BOOL GetProviderFlags(
   DWORD dwProviderId,
   LPDWORD lpdwFlags
);
```

The **GetProviderFlags** method returns the capability flags associated with a provider.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpdwFlags

A pointer to an unsigned integer which receives one or more bit flags when the method returns. If there are no capabilities defined for the provider, this value will be set to LLM_PROVIDER_FLAG_NONE. This parameter cannot be NULL and does not need to be initialized by the caller.

Name	Description
LLM_PROVIDER_FLAG_NONE	No provider requirements specified
LLM_PROVIDER_FLAG_REQUIRE_TLS	Requires a secure TLS (HTTPS) connection
LLM_PROVIDER_FLAG_REQUIRE_AUTH	Requires authentication (e.g., API key or token)
LLM_PROVIDER_FLAG_API_VERSION	Requires an explicit API version parameter
LLM_PROVIDER_FLAG_MODEL_LIST	Supports enumerating available models
LLM_PROVIDER_FLAG_MODEL_INFO	Supports retrieving detailed model information

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetProviderFlags** method retrieves the capability flags associated with the specified provider. These flags describe the features and requirements supported by the provider, such as whether authentication is required or if model enumeration is available.

The value returned in ***lpdwFlags*** may contain one or more LLM_PROVIDER_FLAG values combined with a bitwise OR operation. If no capabilities are defined for the provider, the value will be LLM_PROVIDER_FLAG_NONE.

This method does not require an active client session and can be used to query provider capabilities before establishing a connection. This can be useful when determining how to configure a session or which features are available for a given provider.

If ***dwProviderId*** specifies an invalid provider, the method fails and returns zero.

Example

```
DWORD dwFlags = LLM_PROVIDER_FLAG_NONE;
```

```
if (!llmClient.GetProviderFlags(dwProviderId, &dwFlags))
{
    _tprintf(_T("Unable to get provider flags\n"));
    return;
}

if (dwFlags & LLM_PROVIDER_FLAG_REQUIRE_AUTH)
{
    _tprintf(_T("Provider requires authentication\n"));
}

if (dwFlags & LLM_PROVIDER_FLAG_MODEL_LIST)
{
    _tprintf(_T("Provider supports model enumeration\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetProviderInfo, GetProviderId, GetProviderName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetProviderId Method

```
DWORD GetProviderId();

DWORD GetProviderId(
   LPCTSTR lpszProviderName,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName
);
```

The **GetProviderId** method returns the current provider ID or one based on a provider name, endpoint URL or model name.

Parameters

lpszProviderName

A null-terminated string which specifies the provider name. This parameter may be NULL if you do not wish to match against a provider name. At least one parameter must be specified. If a provider name is specified, it is given precedence over the other parameters.

lpszBaseUrl

A null-terminated string which specifies an endpoint URL. This parameter may be NULL if you do not wish to match against an endpoint URL. At least one parameter must be specified. An endpoint URL will only be used if the function is unable to determine the provider ID from the *lpszProviderName* parameter. If an endpoint URL is specified, it is given precedence over a model name.

lpszModelName

A null-terminated string which specifies a model name. This parameter may be NULL if you do not wish to match against a model name. At least one parameter must be specified. A model name will only be used if the function is unable to determine the provider ID from the *lpszProviderName* or *lpszBaseUrl* parameters.

Return Value

If the method succeeds, the return value is the provider ID. If the method fails, the return value is LLM_PROVIDER_UNKNOWN. To get extended error information, call **GetLastError**.

Remarks

The **GetProviderId** method can be called in two different ways. If the method is called without any arguments, it will return the provider ID associated with the current client session. This is useful when an application supports multiple providers and needs to adjust its behavior based on the active provider.

Another version of this method attempts to determine the provider ID using the parameters passed to it in order of precedence. If a provider name is specified and recognized, that value is considered authoritative and the remaining parameters are ignored. If the provider name is not recognized, the method will attempt to determine the provider from the endpoint URL. If the provider cannot be determined from the endpoint URL, the method will attempt to infer the provider from the model name.

The method performs normalization on the input values before matching them against known providers. Leading and trailing whitespace is ignored, provider names are matched without regard to case and common variations of provider names are recognized.

When matching against a model name, the method identifies the company or organization most commonly associated with the model family. This may not reflect the actual service endpoint hosting the model. For example, a model hosted by a local inference server or aggregate provider may still be identified as the company who originally created the model.

This method only recognizes well-known providers, endpoint URLs and model families. If the provider cannot be determined, the method will return `LLM_PROVIDER_UNKNOWN` and the last error code will be set to `ST_ERROR_PROVIDER_NOT_FOUND`.

Example

```
// Get the provider ID for the current client session
DWORD dwProviderId = llmClient.GetProviderId();

if (dwProviderId != LLM_PROVIDER_UNKNOWN)
{
    TCHAR szBaseUrl[128] = {0};
    llmClient.GetProviderUrl(szBaseUrl, _countof(szBaseUrl));

    _tprintf(_T("Provider ID: %lu\n"), dwProviderId);
    _tprintf(_T("Base URL: %s\n"), szBaseUrl);
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Unable to get provider information, error 0x%08lx\n"), dwError);
}

// Determine the provider ID using a base endpoint URL
LPCTSTR lpszUrl = _T("https://api.openai.com/v1/")

dwProviderId = llmClient.GetProviderId(NULL, lpszUrl, NULL);

if (dwProviderId != LLM_PROVIDER_UNKNOWN)
{
    TCHAR szName[64] = {0};
    llmClient.GetProviderName(dwProviderId, szName, _countof(szName));

    _tprintf(_T("The provider for %s is %s\n"), lpszUrl, szName);
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Unable to get the provider ID, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProviderFlags, GetProviderInfo, GetProviderName, GetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetProviderInfo Method

```
BOOL GetProviderInfo(
   DWORD dwProviderId,
   LPLLM_PROVIDER_INFO lpProviderInfo
);
```

The **GetProviderInfo** method returns information about the specified service provider type.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpProviderInfo

A pointer to an LLM_PROVIDER_INFO structure which receives information about the specified provider. The dwSize member must be initialized to the size of the structure before calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetProviderInfo** method returns general information about a provider, including its display name, description, default host name, default model and related configuration values.

The *lpProviderInfo* parameter must point to an LLM_PROVIDER_INFO structure initialized by the caller. The dwSize member must be set to the size of the structure. This enables the method to validate the structure version before returning provider information.

This method does not require an active client session. It can be used to query provider information before creating a session or connecting to a provider.

The information returned by this method describes the provider defaults defined by the library. A client session may use different values if the application overrides the provider, host name, base URL, model name or other session settings.

Example

```
DWORD dwProviderId = LLM_PROVIDER_NONE;
TCHAR szBaseUrl[MAX_PATH] = {0};

// Determine the provider ID for the current session and get
// the base URL used for API requests
dwProviderId = llmClient.GetProviderId(szBaseUrl, _countof(szBaseUrl));

if (dwProviderId == LLM_PROVIDER_NONE)
{
    _tprintf(_T("Unable to get provider ID for this session\n"));
    return;
}

// Get the current client options
DWORD dwOptions = LLM_OPTION_NONE;
```

```
if (!llmClient.GetOptions(&dwOptions))
{
    _tprintf(_T("Unable to get client options for this session\n"));
    return;
}

// Get information about the current provider
LLM_PROVIDER_INFO providerInfo = { sizeof(providerInfo), 0 };

if (!llmClient.GetProviderInfo(dwProviderId, &providerInfo))
{
    _tprintf(_T("Unable to get provider information\n"));
    return;
}

_tprintf(_T("Connected to %s using %s, the connection is %s\n"),
         providerInfo.szDisplayName,
         szBaseUrl,
         ((dwOptions & LLM_OPTION_SECURE) ? _T("secure") : _T("not secure")));
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetProviderFlags, GetProviderId, GetProviderName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetProviderName Method

```
INT GetProviderName(
   DWORD dwProviderId,
   LPTSTR lpszProviderName,
   INT nMaxLength
);

INT GetProviderName(
   DWORD dwProviderId,
   CString &strProviderName
);
```

The **GetProviderName** method returns the display name of the specified provider.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpszProviderName

A pointer to a string buffer which will contain the display name for the specified provider when the method returns. This parameter cannot be NULL and must be large enough to contain the display name, including the terminating null character. If the buffer is not large enough to contain the entire name, it will be truncated. An alternate version of this method accepts a **CString** object which will contain the provider name when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. This parameter cannot be zero and the method will fail if an invalid size is provided by the caller.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetProviderName** method returns the display name associated with the specified provider. This is a human-readable string which can be used for logging, status messages or user interface display.

This method is a convenience wrapper which returns the provider display name without requiring the caller to retrieve the full provider information structure using **GetProviderInfo**.

If the buffer specified by ***lpszProviderName*** is not large enough to contain the entire provider name, the string will be truncated. The value returned by the method indicates the number of characters copied to the buffer, not including the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetProviderFlags, GetProviderId, GetProviderInfo, GetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetProviderUrl Method

```
INT GetProviderUrl(
   LPCTSTR lpszBaseUrl,
   INT nMaxLength
);

INT GetProviderUrl(
   CString &strBaseUrl
);
```

The **GetProviderUrl** method returns the current endpoint URL used for client requests.

Parameters

lpszBaseUrl

A pointer to a buffer that receives the base URL used by the client session when making requests to the provider. This parameter may be NULL if the base URL is not required. An alternate version of this method accepts a **CString** object which will contain the URL when the method returns.

nMaxLength

The size of the buffer specified by the lpszBaseUrl parameter, in characters. If lpszBaseUrl is NULL, this value should be zero. If a buffer is provided, it must be large enough to store the base URL, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the base URL, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetProviderUrl** method can be used to determine the base URL path used by the provider associated with the current client session.

The base URL is returned as a null-terminated string. If the buffer is not large enough to receive the entire string, the method fails and sets the error code to ST_ERROR_BUFFER_TOO_SMALL. We recommend a buffer size of at least 256 characters when retrieving the base URL.

To determine the default endpoint URL used by a specific service provider, use the **GetDefaultUrl** method.

Example

```
DWORD dwProviderId = llmClient.GetProviderId();

if (dwProviderId != LLM_PROVIDER_UNKNOWN)
{
    TCHAR szBaseUrl[256] = {0};
    llmClient.GetProviderUrl(szBaseUrl, _countof(szBaseUrl));

    _tprintf(_T("Provider ID: %lu\n"), dwProviderId);
    _tprintf(_T("Base URL: %s\n"), szBaseUrl);
}
else
{
    DWORD dwError = llmClient.GetLastError();
```

```
    _tprintf(_T("Unable to get provider information, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultUrl, GetProviderFlags, GetProviderId, GetProviderInfo, GetProviderName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetSession Method

```
LPLLM_SESSION GetSession();
```

The **GetSession** method returns the session structure allocated for the current instance of the class.

Parameters

None.

Return Value

This method returns a pointer to the LLM_SESSION structure that was allocated for the class instance. If no session structure has been created, it will return NULL.

Remarks

The **GetSession** method returns a pointer to the LLM_SESSION structure allocated by the **CreateSession** method and used by the **ConnectEx** method to establish a connection. An session structure is used when the application requires more precise control over how the connection to the service provider is established and what parameters are used when interacting with the model.

It is important to note that the session created by the **CreateSession** method is owned and managed by the class instance. Although you may use this method to access the structure members, you should not modify them. The memory allocated for the structure will be automatically released when the class destructor is called. You can also call the **DeleteSession** method if you wish to free the structure immediately, although this is not necessary.

If your project is built using the Unicode character set, this method will return a structure where the string members will point to wide (UTF-16) characters. If your project uses the Multi-Byte character set, the string members will point to UTF-8 encoded strings.

If the **CreateSession** method has not been called, this method will return NULL.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSession, ConnectEx, DeleteSession, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetStatusCode Method

```
INT GetStatusCode();
```

The **GetStatusCode** method returns the most recent HTTP status code for a client request.

Parameters

None.

Return Value

If the method succeeds, the return value is the HTTP status code. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatusCode** method returns the HTTP status code returned by the provider for the most recent request associated with the specified client session. This value reflects the result of the underlying HTTP operation used to communicate with the provider.

HTTP status codes are three-digit numeric values which indicate whether a request was successful or if an error occurred. They may be interpreted using the following general ranges:

Value	Description
100-199	Informational responses. These are rarely used in practice and indicate that the request has been received and processing is continuing.
200-299	Successful responses. The request was successfully processed by the provider. A value of 200 (OK) is typically returned for successful API requests.
300-399	Redirection responses. These indicate that additional action may be required to complete the request. These responses are uncommon when using API endpoints.
400-499	Client errors. The request was invalid or could not be processed by the provider. This may indicate issues such as invalid parameters, missing authentication credentials or rate limiting.
500-599	Server errors. The provider encountered an error while processing the request. These errors are typically transient and may succeed if the request is retried.

This method can be used in conjunction with the return value of a method such as **SendMessage** or **Connect**. The status code may provide additional information about why a request was rejected or processed with an error condition by the provider. For example, if a request fails and the status code is 503, this indicates the service is temporarily unavailable.

HTTP status codes are returned by the provider service and their exact meaning may vary slightly between providers. Some providers may also return additional diagnostic information in the response body which can help identify the specific cause of the error. Here are some common error status codes and what they could mean in the context of the client session:

- **400 Bad Request** - The request was malformed or contained invalid parameters. This usually indicates an internal error in the library where a request was not composed correctly. This could occur if the service provider does not support certain functionality, an invalid option was used, or an incorrect endpoint was specified.
- **401 Unauthorized** - The API key is missing, invalid, or expired. If you are using an environment

variable or Windows credential, check to make sure it is defined and contains the correct value. If you continue to experience this error, check the provider account to ensure the key has not been revoked or replaced.

- **403 Forbidden** - The request was understood, but the provider refused to process it. This can occur if the account does not have permission to use the selected model, endpoint, organization, or project. Verify that the API key has access to the requested resource and that billing or account restrictions are not preventing the request.
- **404 Not Found** - The requested resource could not be found. This commonly indicates an incorrect endpoint URL, deployment name, model name, or provider-specific path. Verify that the provider, base URL, and model name are correct for the account being used.
- **408 Request Timeout** - The provider did not complete the request before the timeout expired. This can occur because of network latency, a temporary service issue, a large prompt, or a model that requires more time to generate a response. Consider increasing the timeout value, reducing the prompt or history size, or retrying the request later.
- **409 Conflict** - The request conflicts with the current state of the service or resource. This is uncommon for simple message requests, but may occur with provider-specific operations, duplicate requests, or incompatible settings. Review the request options and retry the operation after any pending provider-side operation has completed.
- **413 Payload Too Large** - The request body is larger than the provider allows. This usually means the prompt, conversation history, attached content, or generated request JSON exceeds the model or endpoint limit. Reduce the prompt size, clear or shorten the conversation history, or select a model with a larger context size. The **TrimHistory** method can be used to reduce the amount of conversation history included with a prompt.
- **415 Unsupported Media Type** - The provider does not support the request content type or encoding. This can indicate that the request was not submitted as JSON, that the wrong endpoint was used, or that the provider does not support the submitted content format. Verify the endpoint and make sure the request format matches the provider requirements.
- **422 Unprocessable Entity** - The provider accepted the request format, but could not process the request contents. This can occur if a model option, reasoning level, response format, tool definition, or other provider-specific setting is not valid for the selected model. Review the selected model and options, and disable unsupported features.
- **429 Too Many Requests** - The provider rejected the request because a rate limit, usage limit, or account quota was exceeded. Applications should wait before retrying the request, reduce the request rate, and avoid retrying immediately in a tight loop. If the error persists, review the provider account limits, billing status, and usage quotas.
- **500 Internal Server Error** - The provider encountered an unexpected server-side error while processing the request. This usually indicates a temporary problem with the provider rather than the application. Retry the request later, and check the provider status page if the problem continues.
- **502 Bad Gateway** - A gateway or proxy received an invalid response from an upstream service. This is typically a temporary provider-side or network condition. Retry the request after a short delay, and check whether the provider is experiencing a service disruption.
- **503 Service Unavailable** - The service is temporarily unavailable because of maintenance,

overload, or resource limits. Retry the request after a delay. If the provider returns a retry interval, the application should wait at least that long before trying again.

- **504 Gateway Timeout** - A gateway or proxy timed out waiting for an upstream service to complete the request. This can occur when the provider is under heavy load or when a response takes too long to generate. Retry the request later, reduce the prompt size, or request a shorter response.

If no request has been made, or if the client has not yet received a response from the provider, the return value may be zero. If the client handle is invalid or a request is currently in progress, the method will return LLM_ERROR (-1). In this case, a status code is not available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Connect, GetLastError, SendMessage, TrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetSystemPrompt Method

```
INT GetSystemPrompt(
   LPTSTR lpszSystemPrompt,
   INT nMaxLength
);

INT GetSystemPrompt(
   CString &strSystemPrompt
);
```

The **GetSystemPrompt** method returns the current system prompt for a client session.

Parameters

lpszSystemPrompt

A pointer to a string buffer which receives the system prompt for the current session. If this parameter is NULL, the method returns the length of the system prompt without copying any characters. If a buffer is provided, it must be large enough to contain the entire prompt, including the terminating null character, or the method will fail. An alternate version of this method accepts a **CString** object which will contain the system prompt when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. If the ***lpszSystemPrompt*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetSystemPrompt** method retrieves the system prompt associated with the specified client session. The system prompt provides initial instructions to the model and influences how responses are generated for subsequent requests.

To retrieve the system prompt, an application can first call this method with a NULL buffer to determine the required length, then allocate a buffer of sufficient size and call the method again to retrieve the prompt.

If a system prompt has not been defined for the session, the method returns zero and the last error code is set to NO_ERROR. This condition indicates that no prompt is currently set and should not be treated as a failure.

The system prompt may be specified when creating a session using **ConnectEx** and the LLM_SESSION structure, or modified at runtime using **SetSystemPrompt**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectEx, SetSystemPrompt, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTimeout** method returns the current timeout value for the specified client session. This value specifies the maximum amount of time, in seconds, that the client will wait for a response from the provider before the request is aborted.

The timeout applies to network operations performed when communicating with the provider. If the timeout period is exceeded, the request will fail and the last error code will indicate a timeout condition.

The timeout value may be specified when creating a session using **ConnectEx** and the LLM_SESSION structure, or modified at runtime using **SetTimeout**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

ConnectEx, GetCacheTime, GetIdleTime, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetTokenUsage Method

```
BOOL GetTokenUsage(
   LPLLM_TOKEN_USAGE lpUsage
);
```

The **GetTokenUsage** method returns token usage statistics for the client session.

Parameters

lpUsage

A pointer to an LLM_TOKEN_USAGE structure which receives token usage statistics for the client session. The dwSize member must be initialized to the size of the structure before calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetTokenUsage** method returns token usage statistics for the specified client session. This information includes token counts reported by the provider, as well as values maintained by the library for the current conversation history.

The ***lpUsage*** parameter must point to an LLM_TOKEN_USAGE structure initialized by the caller. The dwSize member must be set to the size of the structure. This enables the method to validate the structure version before returning usage information.

Token usage values are intended to help applications monitor request size, response size and context usage. The exact values reported may vary by provider and model, and should be treated as informational rather than as a guarantee of billing or model behavior.

This method returns the usage information currently associated with the client session. The values are updated as messages are sent and responses are received.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHistory, GetHistorySize, ResetSession, SendMessage, LLM_TOKEN_USAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::GetUserAgent Method

```
INT GetUserAgent(
   LPTSTR lpszUserAgent,
   INT nMaxLength
);
```

The **GetUserAgent** method returns the current user-agent string used by the client.

Parameters

lpszUserAgent

A pointer to a string buffer which receives the user-agent string for the current session. If this parameter is NULL, the method returns the length of the user-agent string without copying any characters. If a buffer is provided, it must be large enough to contain the entire string, including the terminating null character, or the method will fail.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the ***lpszUserAgent*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetUserAgent** method retrieves the user-agent string currently configured for the client session. This value is included in HTTP requests sent to the provider unless disabled using the LLM_OPTION_NOUSERAGENT option.

If no user-agent string has been explicitly set using **SetUserAgent**, the method may return a default value defined by the library, or no value depending on the current configuration.

To determine the required buffer size, call this method with ***lpszUserAgent*** set to NULL. The return value specifies the number of characters in the string, excluding the terminating null character. If there is no user-agent string defined for the session, the method returns zero and the last error value is cleared (set to NO_ERROR). This condition should not be treated as a failure.

In most cases, the user-agent string will be relatively small. A buffer of 256 characters is typically sufficient for standard user-agent values. If a custom user-agent string is used, a larger buffer (such as 512 characters) may be more appropriate. Applications that require complete flexibility should query the required length and allocate the buffer dynamically.

The user-agent string is typically used by providers for logging or request identification. Developers should avoid including sensitive information in this value.

Example

```
TCHAR szUserAgent[256] = _T("");

if (llmClient.GetUserAgent(szUserAgent, _countof(szUserAgent)) > 0)
{
    _tprintf(_T("User-Agent: %s\n"), szUserAgent);
```

```
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Error retrieving user-agent (0x%08lX)\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetOptions, GetLastError, SetOptions, SetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ImportHistory Method

```
BOOL ImportHistory(
   LPCTSTR lpszFileName,
   LONG nMessageCount,
   LPLONG lpnImported
);

BOOL ImportHistory(
   LPCTSTR lpszFileName,
   LPLONG lpnImported
);
```

The **ImportHistory** method imports a conversation history from a JSON-formatted file.

Parameters

lpszFileName

A pointer to a null-terminated string which specifies the name of the import file. This parameter cannot be NULL or a zero-length string.

nMessageCount

The number of the most recent messages and responses which should be imported. If this value is zero, all messages in the file are imported.

lpnImported

A pointer to an integer which will receive the number of messages which were imported when the method returns. If this information is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ImportHistory** method imports a previously exported conversation history file and restores it as the active conversation history for the client session. Imported messages and responses become part of the current session history and are treated the same as messages generated during the current session.

Importing conversation history replaces the existing history associated with the client session. This method does not merge imported messages with the current conversation history or append them to the existing session. It is not necessary for your application to call **ClearHistory** prior to calling this method.

If the *lpszFileName* parameter specifies a directory or device name, the method will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters, they will be removed. The file must exist and the current process must have permission to read the file.

If the *nMessageCount* parameter is non-zero, only the most recent messages and responses in the import file are restored. If the specified count exceeds the number of available messages in the file, all available messages are imported. If the total number of messages to be imported exceeds the current history size, the method will only import the most recent messages up to that limit. For example, if the import file contains 20 messages and the current history size is 15, the most recent 15 messages will be imported.

The import file must contain valid UTF-8 encoded JSON data in a format compatible with the **ExportHistory** method. If the file does not appear to contain a valid conversation history, the import operation fails before the current session history is cleared.

After the JSON data has been successfully parsed, the current session history is cleared before the imported messages and responses are added to the client session. If an error occurs while processing imported messages or responses, the previous conversation history cannot be restored and partial import data may remain in the current session.

If the import file contains malformed message or response data, the **GetLastError** method will return `ST_ERROR_INVALID_MESSAGE_FORMAT`.

Because imported history files may contain sensitive or private information, applications should take reasonable precautions to protect them. Applications which allow users to import conversation history should consider warning users before importing untrusted or unknown files.

Example

```
LONG nImported = 0;

// Replace the current conversation history with the
// messages stored in the import file
if (llmClient.ImportHistory(
                    _T("%USERPROFILE%\\Documents\\conversation.json"),
                    &nImported))
{
    _tprintf(_T("Imported %ld messages\n"), nImported);
}
else
{
    DWORD dwError = llmClient.GetLastError();
    _tprintf(_T("Unable to import conversation history, error 0x%08lX\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHistory, ExportHistory, GetMessageCount, SetHistorySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero. To get extended error information, call **GetLastError**.

Remarks

A blocking operation may allow the application to be re-entered. For example, the user may press a button while another request is still in progress. If this causes another blocking method to be called for the same client session, the second operation will fail because only one blocking operation can be active at a time.

The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action, such as warning the user that they must wait for the current operation to complete.

When a message is sent to the server using the **SendMessage** or **SendMessageEx** methods, the client will block until the message is sent and the server has responded, or until a timeout has occurred. If you send a request to the server in your application's main UI thread, you should present some visual indicator that the application is waiting for the operation to complete. If you need to ensure your application remains responsive while waiting for a response from the server, you can send messages using a background worker thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Cancel, IsConnected, SendMessage, SendMessageEx, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method verifies that the client session remains valid.

Parameters

None.

Return Value

If the client session is valid and connected to a server, the method returns a non-zero value. If the client is not connected, the method returns zero. To get extended error information, call **GetLastError**.

Remarks

The **IsConnected** method determines whether there is a logical connection to the service provider and the client has successfully authenticated with the API server. Calling this method is not simply a passive status check. The network connection to the provider may be transient and can be automatically re-established if the client session has been idle. Calling **IsConnected** may cause the library to reconnect and re-authenticate with the server if necessary.

If the client must re-authenticate, it will attempt to reconnect to the server and validate the selected model for the session. If the server reports an error condition, such as an invalid API key, the connection will not be re-established and the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Connect, ConnectEx, Disconnect, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

CLlmClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::NormalizeText Method

```
INT NormalizeText(
   LPCTSTR lpszInput,
   INT cchInput,
   LPTSTR lpszOutput,
   INT nMaxLength
);

INT NormalizeText(
   LPCTSTR lpszInput,
   CString &strOutput
);
```

The **NormalizeText** method normalizes message text by removing extraneous whitespace and applying consistent formatting.

Parameters

lpszInput

A pointer to a null-terminated string which specifies the text which should be normalized. This parameter cannot be NULL.

cchInput

An integer which specifies the length of the input string. If this value is -1 the entire input string up to the terminating null character will be normalized.

lpszOutput

A pointer to a string buffer which will contain the normalized text when the method returns. The output string will always be null-terminated. If the string buffer is not large enough to contain all of the normalized text, the method will fail and the last error code will be set to `ST_ERROR_BUFFER_TOO_SMALL`. An alternate version of this method accepts a **CString** object which will contain the normalized text when the method returns.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the output string buffer. This value must be large enough to include a terminating null character. If this value is too small, or an invalid length is specified, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the output buffer, not including the terminating null character. If the method fails, the return value is zero and the **GetLastError** method will return a non-zero value.

Remarks

The **NormalizeText** method processes the input string, decomposing Unicode characters to their canonical equivalents, replacing confusable homoglyphs with normalized characters, removing combining characters, normalizing end-of-line characters, and trimming leading and trailing whitespace including non-breaking spaces and other special Unicode space characters. Although removing unnecessary whitespace may slightly reduce token usage, the primary benefit is producing cleaner and more consistent input for the model to process.

Line break characters are normalized to use Windows end-of-line conventions. Isolated carriage return or line feed characters, Unicode line and paragraph separators, and form feed characters are

converted to CRLF sequences. Multiple consecutive line breaks are reduced to improve consistency.

Normalization can be useful when comparing prompts, checking for empty or whitespace-only input, or applying simple validation rules before a request is submitted. It does not attempt to determine whether the text is safe, appropriate, or semantically valid. Applications should perform any additional validation required for their own use case.

Using this method is optional. Prompt text does not need to be normalized before it is sent to a provider, and using this method is not a required sanitization step. If preserving the exact formatting of the original text is important, such as indentation, tables, or code samples, the application should send the original text instead.

The ANSI version of this method expects the input to be UTF-8 encoded, but will fall back to using the current locale if invalid UTF-8 characters are encountered in the string. The output from this function will always be valid Unicode text. For applications using the ANSI version of this method, this means the output string will use UTF-8 encoding.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EstimateTokenCount, SendMessage, SendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ResetSession Method

```
BOOL ResetSession();
```

The **ResetSession** method resets the current client session to its initial state, clearing any existing conversation context.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ResetSession** method resets the client session to its initial state. Any active connection to the service provider is closed, the conversation history is cleared, and token usage statistics for the session are reset.

This method can be used to discard the current conversation context and start a new session using the existing client handle. Configuration settings such as the selected provider, model name, system prompt, and client options are preserved.

If a blocking operation is currently in progress, this method will fail. Applications should call **IsBlocking** to determine if the client is busy before attempting to reset the session.

Because this method clears the conversation history, it also invalidates any string pointers previously returned in LLM_RESPONSE structures or by message history methods. Applications must not use those pointers after this method returns. If the application needs to keep any message or response text, it should make its own copy before resetting the session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

ClearHistory, Disconnect, TrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SendMessage Method

```
BOOL SendMessage(
   LPCTSTR lpszPrompt,
   LPTSTR lpszResponse,
   LPDWORD lpdwLength,
   LPDWORD lpdwMessageId
);
```

The **SendMessage** method sends a prompt to the model and returns the response as a string.

Parameters

lpszPrompt

A pointer to a null-terminated string which specifies the message prompt which should be sent to the model. This parameter cannot be NULL or a zero-length string. If the string consists only of whitespace characters, or contains invalid Unicode or non-text data, this method will fail.

lpszResponse

A pointer to a string buffer which will contain the model's response to the prompt. The response will always be null terminated and if the buffer is not large enough to store the complete response, the response will be truncated. The ***lpdwLength*** parameter must be initialized with the maximum number of characters which can be copied into the buffer or the method will fail.

lpdwLength

A pointer to an unsigned integer which should be initialized to the size of the ***lpszResponse*** string buffer. When the method returns, this value will be updated with the actual number of characters copied into the buffer, not including the terminating null character. This parameter cannot be NULL and its initial value must be greater than zero.

lpdwMessageId

A pointer to an unsigned integer which will contain the ID for the message when the method returns. This parameter may be NULL if the application does not require the message ID.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method provides a simplified interface for sending a single text prompt to the model and receiving the response as a string. This method is intended for applications that do not require structured input or detailed control over message formatting.

This method will not return until the request has completed, an error has occurred, or the operation has timed out. Applications that call this method from a user interface thread should provide visual feedback while waiting for the response, or perform the request on a background thread to maintain responsiveness.

Applications that accept user input should apply appropriate validation, filtering, or normalization before sending requests to the service provider. The **ValidateText** method verifies that the prompt contains valid message text and the **NormalizeText** method can be used to normalize text formatting. These functions do not perform any semantic analysis of the prompt and do not provide protection against malicious or intentionally misleading input.

This method does not validate or restrict the content of the input message. It is the responsibility of the application to ensure that user-provided input is appropriate for its intended use. This includes considering scenarios where input may attempt to override instructions, inject unintended prompts, or otherwise influence the behavior of the model.

The response is written to the buffer specified by ***lpszResponse***. If the buffer is not large enough to contain the complete response, the text will be truncated. The ***lpdwLength*** parameter will always contain the number of characters actually copied to the buffer.

Line breaks in the response text are normalized before being stored in the client history. Line feed and carriage return characters are converted to the standard Windows CRLF end-of-line sequence, ensuring consistent formatting across different providers and applications. This also helps ensure compatibility with Windows common controls and other user interface components.

Some models are capable of performing additional internal reasoning before generating a response. The amount of reasoning performed and whether it is exposed by the provider is model-specific. The library returns the final response intended for the user and does not currently expose model reasoning or intermediate processing steps. This behavior ensures consistent results across providers, regardless of how individual models implement or expose reasoning capabilities.

If the ***lpdwMessageId*** parameter is specified, it will receive a unique identifier for the message within the current client session. Message identifiers are assigned sequentially, but applications should treat them as opaque values. They are only guaranteed to be unique within a single session and should not be compared across different client sessions.

This method automatically adds the message to the conversation history for the session. Subsequent calls will include previous messages as context, subject to any configured history or token limits. To clear the conversation context, call **ClearHistory** or **ResetSession**.

Applications that require more advanced features, such as structured messages, role-based input, or access to detailed response metadata, should use the **SendMessageEx** method instead.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    CLlmClient llmClient;

    // Establish a connection to OpenAI using the default endpoint
    // and a lightweight model
    BOOL bConnected = llmClient.Connect(
        LLM_PROVIDER_OPENAI,
        LLM_OPTION_DEFAULT,
        LLM_TIMEOUT,
        NULL,                   // Use default base URL
        _T("gpt-5.4-nano"),     // Model name
        _T("%OPENAI_API_KEY%")  // Provider API key
    );

    if (!bConnected)
    {
        dwError = llmClient.GetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        return 1;
    }
```

```
    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse); // number of TCHARs
    DWORD dwMessageId = 0;

    // Send a simple message to the model
    if (llmClient.SendMessage(
                    _T("What is the capital city of Denmark?"),
                    szResponse,
                    &dwResponseLength,
                    &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
    else
    {
        dwError = llmClient.GetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Close the connection explicitly
    llmClient.Disconnect();
    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHistory, GetMessageById, GetTokenUsage, NormalizeText, ResetSession, SendMessageEx, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SendMessageEx Method

```
BOOL SendMessageEx(
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **SendMessageEx** method sends a structured message to the model and returns a structured response.

Parameters

lpMessage

A pointer to a LLM_MESSAGE structure which specifies the message to be sent to the server. This parameter cannot be NULL. The structure must be initialized correctly, with the ***dwSize*** member set to the size of the structure. Unused members of the structure should be initialized to zero or NULL.

lpResponse

A pointer to a LLM_RESPONSE structure which will be populated with information about the response from the server when the method returns. This parameter cannot be NULL. The structure must be initialized correctly, with the ***dwSize*** member set to the size of the structure and all other members set to a value of zero or NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SendMessageEx** method provides the structured form of **SendMessage**. It should be used by applications that need to specify additional message properties, inspect detailed response metadata, or preserve information such as response identifiers, token usage, status values, elapsed time, and the actual model name returned by the provider.

This method performs a blocking operation. It will not return until the request has completed, an error has occurred, or the operation has timed out. Applications that call this method from a user interface thread should provide visual feedback while waiting for the response, or perform the request on a background thread to maintain responsiveness.

The ***lpMessage*** and ***lpResponse*** structures must be initialized before calling this method. At a minimum, the ***dwSize*** member of each structure must be set to the size of the structure, and all unused members should be initialized to zero or NULL.

If a message identifier is returned in the response structure, it is unique within the current client session. Message identifiers are assigned sequentially by the library, but applications should treat them as opaque values. They are not guaranteed to be globally unique across multiple sessions or between different clients.

This method automatically adds the message and response to the conversation history for the session. Subsequent calls may include previous messages as context, subject to any configured history or token limits. To clear the conversation context, call **ClearHistory** or **ResetSession**.

Applications that accept user input should apply appropriate validation, filtering, or normalization before sending requests to the service provider. The **ValidateText** method verifies that the prompt contains valid message text and the **NormalizeText** method can be used to normalize text

formatting. These methods do not perform any semantic analysis of the prompt and do not provide protection against malicious or intentionally misleading input.

Line breaks in the response text are normalized before being stored in the client history. Line feed and carriage return characters are converted to the standard Windows CRLF end-of-line sequence, ensuring consistent formatting across different providers and applications. This also helps ensure compatibility with Windows common controls and other user interface components.

Some models are capable of performing additional internal reasoning before generating a response. The amount of reasoning performed and whether it is exposed by the provider is model-specific. The library returns the final response intended for the user and does not currently expose model reasoning or intermediate processing steps. This behavior ensures consistent results across providers, regardless of how individual models implement or expose reasoning capabilities.

The string members of the LLM_MESSAGE and LLM_RESPONSE structures are managed by the library. When a message is sent, any string pointers provided in the LLM_MESSAGE structure are copied internally, and the application is not required to preserve or maintain those buffers after the method returns.

String pointers returned in the LLM_RESPONSE structure refer to memory managed by the client session. These values remain valid until the message history is cleared or the session is reset or disconnected. Applications should not attempt to free or modify these strings.

The contents of these strings must be treated as read-only. They are owned by the library, and modifying them may result in access violations or other undefined behavior. If your application needs to retain a string beyond the lifetime of the session or modify its contents, it should create its own copy.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    CLlmClient llmClient;

    // Establish a connection to OpenAI using the default endpoint
    // and a lightweight model
    BOOL bConnected = llmClient.Connect(
        LLM_PROVIDER_OPENAI,
        LLM_OPTION_DEFAULT,
        LLM_TIMEOUT,
        NULL,                   // Use default base URL
        _T("gpt-5.4-nano"),     // Model name
        _T("%OPENAI_API_KEY%")  // Provider API key
    );

    if (!bConnected)
    {
        DWORD dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        return 1;
    }

    LLM_MESSAGE clientMessage;
    ZeroMemory(&clientMessage, sizeof(LLM_MESSAGE));
    clientMessage.dwSize = sizeof(clientMessage);
```

```
    clientMessage.lpszContent = _T("What is the capital city of Denmark?");
    clientMessage.nContentLength = _tcslen(clientMessage.lpszContent);

    LLM_RESPONSE serverResponse;
    ZeroMemory(&serverResponse, sizeof(serverResponse));
    serverResponse.dwSize = sizeof(serverResponse);

    if (llmClient.SendMessageEx(&clientMessage, &serverResponse))
    {
        _tprintf(_T("Flags: 0x%08lX\n"), serverResponse.dwFlags);
        _tprintf(_T("Status: %u\n"), serverResponse.dwStatus);
        _tprintf(_T("Elapsed: %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("Output Tokens: %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("Content Length: %u\n"), serverResponse.nContentLength);
        _tprintf(_T("Response ID: %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("Model Name: %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("Status Text: %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));

        if (serverResponse.nContentLength > 0)
            _tprintf(_T("\n%s\n"), serverResponse.lpszContent);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Close the connection explicitly
    llmClient.Disconnect();
    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHistory, GetMessageById, GetTokenUsage, NormalizeText, ResetSession, SendMessage, ValidateText, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetCacheTime Method

```
BOOL SetCacheTime(
   INT nCacheTime
);
```

The **SetCacheTime** method sets the cache lifetime used by the client for model or provider data.

Parameters

nCacheTime

An integer value which specifies the cache lifetime, in seconds, for provider and model metadata. A value of zero disables caching and forces the client to retrieve this information from the server for each request. Larger values reduce network requests but may result in stale information being used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetCacheTime** method controls how long the client caches provider and model metadata. This includes information such as the list of available models and details about the currently selected model.

Caching reduces the number of network requests required when retrieving model lists or model information, which can improve performance and reduce latency. However, because model availability and capabilities may change over time, cached data may become stale if it is retained for extended periods.

Changing this value will have no effect if the LLM_OPTION_NOCACHE option has been specified.

Setting the cache time to zero disables caching and is equivalent to specifying the LLM_OPTION_NOCACHE option. Note that this differs from specifying a cache time of zero in the LLM_SESSION structure during initialization, where a value of zero indicates that a default cache lifetime will be used.

The cache time is specified in seconds. If a value greater than or equal to 1000 is specified, it is assumed that the value may have been provided in milliseconds, and it will be automatically scaled to seconds. The specified cache time is clamped to an internal minimum and maximum value if it falls outside the supported range. The default cache time is one hour.

In most cases, the default cache time provides a good balance between performance and accuracy. Applications that frequently query model information or require real-time updates may choose to reduce or disable caching.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

EnumModels, GetCacheTime, GetModelInfo, SetIdleTime, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetHeader Method

```
BOOL SetHeader(
   LPCTSTR lpszHeaderName,
   LPTSTR lpszHeaderValue
);
```

The **SetHeader** method sets the value of a custom HTTP header issued with subsequent requests.

Parameters

hClient

A handle to the client session.

lpszHeaderName

A pointer to a null-terminated string which specifies the name of the HTTP header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszHeaderValue

A pointer to a null-terminated string which specifies the value of the HTTP header. Header values cannot contain carriage-return or linefeed characters. If the value contains Unicode characters, they will be UTF-8 encoded. If this parameter or NULL or a zero-length string, the header field is deleted from the request headers.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**. If the header name contains illegal characters or cannot be changed (e.g., attempting to change the `Authorization` header), the method will return zero and the last error code will be set to `ST_ERROR_INVALID_HEADER_NAME`. If the header value contains illegal characters, the last error code will be set to `ST_ERROR_INVALID_HEADER_VALUE`.

Remarks

The **SetHeader** method assigns a custom HTTP request header that will be included with subsequent requests made using the current client session. If the specified header already exists in the custom header list, its value will be replaced. If the header value is an empty string, the header will be removed.

This method is intended for advanced applications that require lower-level access to provider-specific functionality. Most applications should not need to use custom request headers. Some headers are managed internally by the library and cannot be overridden using this method. This includes headers required for authentication, content formatting, message length, host identification, and other protocol-specific functionality required by the current provider.

Examples of headers that are managed internally by the library include:

- `Authorization`
- `Content-Length`
- `Content-Type`
- `Host`
- `User-Agent`

Custom headers remain associated with the current client session until they are removed using the

ClearHeaders method, the **ResetSession** method is called, or the connection is closed using the **Disconnect** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHeaders, Disconnect, GetHeader, ResetSession, SendMessage, SetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetHistorySize Method

```
BOOL SetHistorySize(
   LONG nMaxMessages
);
```

The **SetHistorySize** method sets the maximum number of message exchanges retained in the conversation history.

Parameters

nMaxMessages

An integer value which specifies the maximum number of messages that will be retained in the conversation history for the client session. Older messages will be discarded automatically as new messages are added once this limit is reached. This value cannot be less than zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetHistorySize** method controls the number of messages retained in the conversation history for a client session. This history is used to provide context for subsequent requests to the model.

When the number of messages exceeds the specified limit, the oldest messages are removed automatically. This ensures that the total number of stored messages does not exceed the configured maximum.

Reducing the history size can help limit memory usage and reduce the amount of context sent with each request, which may improve performance and reduce token usage. However, limiting the history size may also reduce the model's ability to maintain context across multiple interactions.

This setting works in conjunction with token limits imposed by the selected model. Even if the maximum number of messages has not been reached, older messages may be excluded from a request if including them would exceed the model's context window.

If ***nMaxMessages*** has a value of zero, conversation history is effectively disabled. In this mode, only the current message and response are retained for the duration of the request, and no prior messages are included as context for subsequent requests. Disabling conversation history may reduce the model's ability to maintain context across multiple interactions and can impact the quality or relevance of responses. Methods such as **GetFirstMessage** and **GetNextMessage** will fail with the last error code set to ST_ERROR_NO_MESSAGE_HISTORY.

Disabling conversation history does not provide additional security or privacy. It only affects whether previous messages are retained and automatically included in subsequent requests. The content of each request is still transmitted to the service provider and may be processed or stored in accordance with that provider's policies. Applications that require secure handling of sensitive information should avoid including confidential data in requests and review the data handling practices of the selected service provider.

Even when message history is disabled, the model may still produce responses that appear contextually relevant within a single interaction. This is because the current request can include sufficient context for the model to generate an appropriate response. However, without retaining previous messages, the model does not have access to prior exchanges in the conversation, and

continuity across multiple requests may be limited. In most cases, retaining a moderate history size provides better results for conversational applications.

The default history size is 100 message exchanges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

ClearHistory, GetHistorySize, ResetSession, SendMessage, SendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetIdleTime Method

```
BOOL LlmSetIdleTime(
   INT nIdleTime
);
```

The **SetIdleTime** method sets the idle timeout period for the client session.

Parameters

nIdleTime

An integer value which specifies the idle timeout period in seconds. A value of zero specifies a reasonable default value should be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetIdleTime** method sets the idle timeout period, in seconds, for the specified client session. This value determines how long the library will retain an open connection between requests before automatically closing it and establishing a new connection when needed.

Setting a lower idle time may result in more frequent reconnections, which can increase overhead but may be desirable in environments where connections are short-lived. Higher values allow connections to be reused for longer periods, which can improve performance when making repeated requests.

Changing this value is only meaningful when the LLM_OPTION_KEEPALIVE option is enabled. If keep-alive is disabled for the session, the connection will not be reused regardless of the idle time setting. If the *nIdleTime* parameter is set to zero, a new connection is established for each request. In most cases, a default idle time of 15 seconds provides a good balance between performance and compatibility.

This setting applies only to the client. It cannot force a server to maintain a persistent connection for a longer period of time. If a connection is closed due to inactivity, the library will automatically reconnect as needed. This behavior is transparent to the application and does not require any additional method calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

IsConnected, GetIdleTime, SetCacheTime, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or LLM_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

GetErrorString, GetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetModelFlags Method

```
BOOL SetModelFlags(
   DWORD dwModelFlags
);
```

The **SetModelFlags** method sets model behavior flags for a client session.

Parameters

dwModelFlags

An unsigned integer which specifies one or more capability flags for the selected model. This value cannot be zero.

Name	Description
LLM_MODEL_FLAG_CHAT	Supports conversational text generation
LLM_MODEL_FLAG_SYSTEM_ROLE	Accepts a system role message in the input
LLM_MODEL_FLAG_STREAMING	Supports streaming responses
LLM_MODEL_FLAG_STRUCTURED	Supports structured output (e.g., JSON or schema)
LLM_MODEL_FLAG_VISION	Supports image inputs (multimodal)
LLM_MODEL_FLAG_TOOLS	Supports tool or function calling
LLM_MODEL_FLAG_REASONING	Supports reasoning-oriented generation
LLM_MODEL_FLAG_RESPONSES	Supports the Responses API (unified endpoint)
LLM_MODEL_FLAG_FUNCTIONS	Supports OpenAI-style function calling with schemas
LLM_MODEL_FLAG_AUDIO	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	Supports image generation
LLM_MODEL_FLAG_VIDEO	Supports video generation
LLM_MODEL_FLAG_TRANSCRIPTION	Supports speech-to-text transcription
LLM_MODEL_FLAG_INSTRUCTIONS	Supports a dedicated instruction field outside message lists
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	Uses `max_completion_tokens` instead of `max_tokens` in requests

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetModelFlags** method overrides the capability flags associated with the currently selected model. These flags determine how the client formats requests and interprets responses for the model.

Model capability flags are normally determined automatically by the library based on the selected provider and model. This method is primarily intended for advanced scenarios where the default behavior must be adjusted, such as when working with custom providers, locally hosted models, or when a model's capabilities have changed and are not yet reflected in the library.

The specified flags must include **LLM_MODEL_FLAG_CHAT**, as the API requires a chat-based interface for sending and receiving messages. Specifying flags that are not supported by the selected model may result in request failures or undefined behavior.

Changing model flags can affect how requests are constructed, such as which parameters are used, how input is formatted, and which features are enabled. Applications should only modify these flags if they fully understand the capabilities of the target model.

If you enable a capability that is not supported by the model or provider, this will often result in an error condition and may cause subsequent requests to fail. For example, if you attempt to force the use of a system prompt by setting the LLM_MODEL_FLAG_SYSTEM_ROLE flag, this can cause unexpected failures on models hosted locally using an application like LM Studio.

Changing the current model using the **SetModelName** method will reset the model flags for the session back to their default values for the new model.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetModelInfo, GetModelFlags, GetModelName, SetModelName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetOptions Method

```
BOOL SetOptions(
   DWORD dwOptions
);
```

The **SetOptions** method changes the current client option flags.

Parameters

dwOptions

An unsigned integer that specifies one or more options.

Value	Description
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be used when a provider does not require a User-Agent header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetOptions** method replaces the current option flags for the client session. These flags control

how the client establishes connections and sends requests to the service provider.

Applications should use care when changing options that affect the underlying connection, such as LLM_OPTION_SECURE or LLM_OPTION_PROXY. Changing these options may require the client to establish a new connection using different settings.

The LLM_OPTION_SECURE option forces the client to use HTTPS when communicating with the provider. This is appropriate for public API providers, but may not be supported by locally hosted providers such as LM Studio or Ollama. Attempting to force a secure connection to a provider that does not support HTTPS will cause the connection to fail.

The LLM_OPTION_PROXY option causes the client to use the configured proxy settings when connecting to the provider. This may be required in managed or corporate network environments, but should not be enabled unless proxy support is actually needed.

The LLM_OPTION_NOUSERAGENT option does not make requests anonymous. The provider may still identify the client based on authentication credentials, IP address information, request patterns, or other HTTP headers included with the request. For more information, refer to the **SetUserAgent** method.

Setting or clearing the LLM_OPTION_NOCACHE option changes whether provider and model metadata may be cached. Applications can also use the **SetCacheTime** method to control caching behavior in seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Connect, ConnectEx, GetCacheTime, GetOptions, GetUserAgent, SetCacheTime, SetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetSystemPrompt Method

```
BOOL SetSystemPrompt(
   LPCTSTR lpszSystemPrompt
);
```

The **SetSystemPrompt** method changes the current system prompt for a client session.

Parameters

hClient

A handle to the client session.

lpszSystemPrompt

A pointer to a string buffer which will contain the new system prompt for the current session. If this parameter is NULL or an empty string, a system prompt will not be included with new requests.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A system prompt provides instructions that are included with each request sent to the model. It can be used to define the expected role, tone, response style, formatting rules, or other application-specific guidance that should apply to the session.

For example, an application might use a system prompt to request short responses, avoid Markdown formatting, return structured text, or answer only within the context of a specific feature or workflow.

Because conversation history is included with each request, messages that were generated using a previous system prompt may still influence subsequent responses. Changing or clearing the system prompt does not retroactively modify existing messages in the history. If the new prompt represents a significant change in behavior, applications may want to clear the history to avoid unintended interactions between the previous and current instructions.

If the new prompt represents a significant change in behavior, applications may want to call **ClearHistory** before sending additional messages.

The system prompt should be treated as guidance, not a guarantee. Different providers and models may interpret system instructions differently, and user messages may still influence the response. If ***lpszSystemPrompt*** is NULL or points to an empty string, the current system prompt is cleared and no system prompt will be included with new requests.

Support for system prompts may vary between providers and models. Some providers treat the system prompt as a distinct message role, while others incorporate it as part of a separate instruction field. The library will map the system prompt to the appropriate format for the selected provider, but the exact behavior and influence on the response may differ. Applications should test system prompt behavior with their selected provider to ensure it produces the desired results.

Note that some providers or models will not provide any support for a system role. This is more common with locally hosted models using applications like LM Studio or Ollama. To determine if a model supports system prompts, use the **GetModelFlags** method and check for the LLM_MODEL_FLAG_SYSTEM_ROLE flag.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHistory, GetModelFlags, GetSystemPrompt, ResetSession, SendMessage, SendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetTimeout Method

```
BOOL SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a blocking operation to complete.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetTimeout** method sets the timeout period, in seconds, for blocking operations such as sending a message to the server. If a request does not complete within the specified time, the operation will fail and return a timeout error.

The minimum timeout period is 5 seconds and the maximum timeout period is 300 seconds. If a value outside this range is specified, it will be clamped to the nearest supported value. Specifying a value of zero resets the timeout to the default value of 60 seconds.

The appropriate timeout value may vary depending on the selected provider, model, and network conditions. Larger or more complex models may require additional time to generate a response. Locally hosted models may also require longer timeouts depending on the available hardware resources, such as CPU or GPU performance.

The maximum timeout period is intended to prevent applications from waiting indefinitely for a response. If an operation consistently requires several minutes to complete, applications should consider performing the request on a background thread or using a smaller or faster model.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

Connect, GetTimeout, SendMessage, SendMessageEx, SetCacheTime, SetIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::SetUserAgent Method

```
BOOL SetUserAgent(
   LPCTSTR lpszUserAgent
);
```

The **SetUserAgent** method changes the current user-agent string used by the client.

Parameters

lpszUserAgent

A pointer to a string buffer which will contain the new user-agent string for the current session. If this parameter is NULL or a zero-length string, the method will reset the user-agent string to a default value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetUserAgent** method changes the user-agent string used by the client session. The specified string is trimmed to remove leading and trailing whitespace and will be used the next time the client establishes a connection to the provider.

If *lpszUserAgent* is NULL or points to an empty string, the current user-agent string is cleared and a new default value is generated for the session.

The default user-agent string uses a format similar to Chromium-based browsers and includes information about the current Windows platform and the SocketTools version. If an application name was specified when the session was created, it may also be included in the default value.

For example, a default user-agent string may look similar to the following:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/134.0.0.0 Safari/537.36
SocketTools/12.0
```

Applications normally do not need to change the user-agent string. However, a custom value may be useful when an application needs to identify itself to a provider, gateway, proxy, or logging system.

Developers should avoid including sensitive information, such as API keys, account identifiers, or user-specific data, in the user-agent string.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetOptions, GetUserAgent, SetOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** method. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::TrimHistory Method

```
LONG TrimHistory(
   LONG nMessageCount,
   LONG nMaxTokens
);
```

The **TrimHistory** method removes older messages from the conversation history for the current client session.

Parameters

nMessageCount

The number of messages to remove from the beginning of the conversation history. If this value is zero, messages are only removed if required by the ***nMaxTokens*** parameter. If this value is equal to or greater than the number of messages in the history, the entire history is cleared.

nMaxTokens

The maximum number of input tokens that should remain in the conversation history. If this value is zero, token usage is not checked. If the current history exceeds this value, additional older messages may be deleted until the total input token count is below the specified limit.

Return Value

If the method succeeds, the return value is the number of messages deleted from the conversation history. If the method fails, the return value is LLM_ERROR. To get extended error information, call **GetLastError**.

If no messages are available to delete, the return value is zero.

Remarks

The **TrimHistory** method removes messages from the beginning of the conversation history. This can be used to reduce memory usage, discard older context, or limit how much previous conversation text is included with future requests.

The ***nMessageCount*** parameter specifies the minimum number of older messages to remove. If ***nMaxTokens*** is greater than zero, the method also checks the total number of input tokens currently stored in the message history. If the token count exceeds the specified limit, additional older messages will be removed until the total input token count is below the specified threshold.

When both parameters are specified, the method may remove more messages than requested by ***nMessageCount*** to reduce the token count below the limit specified by ***nMaxTokens***. The return value always indicates the actual number of messages removed.

If the number of messages to remove is equal to or greater than the number of messages currently stored, the entire conversation history is cleared. This is equivalent to calling **ClearHistory**.

This method only removes the local message and response history maintained by the client session. It does not change the current provider, model, system prompt, connection settings or cached provider metadata.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csllmv12.lib

See Also

ClearHistory, GetHistorySize, SetHistorySize, GetFirstMessage, GetNextMessage, GetTokenUsage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ValidateModel Method

```
BOOL ValidateModel(
   LPCTSTR lpszModelName
);
```

The **ValidateModel** method validates that a model name is recognized or supported by the current provider.

Parameters

lpszModelName

A null-terminated string which specifies the name of the model to validate. This parameter cannot be NULL or a zero-length string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ValidateModel** method verifies that a model name appears to be valid for the current session. Validation may be performed using locally available information or by querying the provider for a list of supported models. If the provider supports returning model information, this method may perform a lookup to confirm that the specified model name is recognized. This may require a network request and can cause the method to block while the model list is retrieved.

If the provider does not support returning model information, the method only performs basic validation of the model name. For example, it will check the name to ensure it does not contain any illegal characters and uses a format that is common for most service providers. This method is most commonly used with applications to validate user input.

A successful return value from this method only indicates that the model name is syntactically valid. It does not guarantee that the selected service provider will accept or allow the model to be used. A provider may reject a valid model name for several reasons, including the model being unavailable for the current account, restricted to specific subscription tiers, region-limited, deprecated, disabled by administrative policy, or not supported by the endpoint being used. Providers may also change model availability over time without notice. Applications should be prepared to handle errors returned when establishing a session or sending a request, even if the model name was validated successfully.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumModels, GetModelInfo, SetModelFlags, SetModelName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient::ValidateText Method

```
BOOL ValidateText(
   LPCTSTR lpszMessageText,
   INT cchMessageText
);
```

The **ValidateText** method performs basic validation on text before it is submitted to a model.

Parameters

lpszMessageText

A pointer to the message text to validate. This parameter cannot be NULL.

cchMessageText

The number of characters in the message text, excluding the terminating null character. If this parameter is omitted or the value is -1, the text must be null-terminated and the entire string is checked.

Return Value

If the message text appears to be valid, the method returns non-zero. If the message text is empty, contains invalid characters, or appears to contain non-text data, the method returns zero. To get extended error information, call **GetLastError**.

Remarks

The **ValidateText** method is intended to help applications detect invalid or malformed input before sending a request to a model. It checks for common problems such as empty strings, text that consists only of spaces or line breaks, illegal Unicode code points, non-printable characters, and input that appears to contain binary data. The message text cannot contain embedded null characters.

Applications that accept messages from users or other external sources should validate the text before submitting it to a model. This can help detect malformed or invalid input before a request is sent, reducing the chance of unexpected results or incurring unnecessary usage costs.

This method does not evaluate the quality, safety, or semantic meaning of the message. Although this method is most commonly used to check prompts sent to a model, it can be used to validate any type of text to ensure it is not malformed and contains printable characters.

If your project uses the Unicode character set, this method validates the text as UTF-16 Unicode. If your project uses the Multi-Byte character set, it attempts to interpret the input as UTF-8 encoded text. If the input is not valid UTF-8, it attempts to convert the text to Unicode using the active code page and the resulting Unicode text is validated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EstimateTokenCount, NormalizeText, SendMessage, SendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Data Structures

- LLM_MESSAGE
- LLM_MODEL_INFO
- LLM_PROVIDER_INFO
- LLM_RESPONSE
- LLM_SESSION
- LLM_TOKEN_USAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_MESSAGE

The **LLM_MESSAGE** structure represents a message sent from the client to the LLM server.

```
typedef struct _LLM_MESSAGE
{
    DWORD   dwSize;
    DWORD   dwRoleId;
    DWORD   dwFlags;
    DWORD   dwReasoningLevel;
    DWORD   dwMessageId;
    DWORD   dwReserved;
    TIME_T  tTimestamp;
    LONG    nMaxTokens;
    LONG    nInputTokens;
    LONG    nContentLength;
    LPCTSTR lpszContent;
} LLM_MESSAGE, *LPLLM_MESSAGE;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_MESSAGE)` before the structure is used.

dwRoleId

The role associated with the message, such as user, assistant or system. This member should be one of the following values:

Constant	Description
LLM_ROLE_UNKNOWN	Unknown or invalid role. This value is typically used when a message has not been initialized correctly or could not be classified.
LLM_ROLE_USER	A message originating from the end-user. This is the most common role and represents input provided by the application or user interface.
LLM_ROLE_ASSISTANT	A response generated by the language model. Messages with this role are typically returned by the library as part of the conversation history.
LLM_ROLE_SYSTEM	A system-level instruction that defines how the model should behave. This is typically used to set context, tone, or constraints for the session before user messages are processed.
LLM_ROLE_FUNCTION	A legacy role used for method call messages in older APIs. This role is retained for compatibility but is generally not used in new implementations.
LLM_ROLE_TOOL	A message generated by an external tool or method invoked by the model. This role is used when integrating tool or method-calling workflows.
LLM_ROLE_TOOLUSER	A user message that is specifically intended for processing by a tool or method, rather than direct model interpretation.

LLM_ROLE_DEFAULT	The default role assigned to messages. This is equivalent to LLM_ROLE_USER.

dwFlags

A value reserved for future use. This member should always be set to zero or `LLM_MESSAGE_FLAG_NONE`.

dwReasoningLevel

The reasoning effort level associated with the message. This member may be set to one of the `LLM_REASONING` constants to control how much reasoning the model should apply when generating a response. A value of `LLM_REASONING_DEFAULT` uses the provider or model default behavior.

dwMessageId

An identifier assigned to the message. For application-supplied input messages, this value is initially zero. When a message has been added to a session or returned from the message history, this member may contain a unique identifier assigned by the library.

dwReserved

A reserved value for future use. This member should always be zero.

tTimestamp

The timestamp associated with the message. For application-supplied input messages, this value is typically zero. When available, this member contains the time the message was created.

nMaxTokens

The maximum number of output tokens associated with the message. This value is primarily used when submitting a request and may be zero to indicate the session or provider default should be used.

nInputTokens

The number of input tokens associated with the message content. This value is typically provided by the library when token usage information is available.

nContentLength

The length of the message content, in characters, not including the terminating null character. If this value is zero, the content length may be determined from the string.

lpszContent

A pointer to a null-terminated string which contains the message text. This member may be NULL if the message consists only of media content.

Remarks

This structure is used to describe a single message in a conversation with a language model. Messages are typically assigned a role using one of the constants, such as `LLM_ROLE_USER` for user input, `LLM_ROLE_ASSISTANT` for a model response, or `LLM_ROLE_SYSTEM` for an instruction message that defines the behavior of the session. In most applications, those roles are required. Other roles are used for advanced scenarios such as tool integration or legacy compatibility.

The **LLM_MESSAGE** structure is used for both input and output operations. When passed to methods such as **SendMessageEx**, it defines a message that will be submitted to the language model. In this case, the application typically initializes only the required members, such as ***dwSize***, ***dwRoleId***, and the content or media fields. When used with methods such as **GetFirstMessage** and **GetNextMessage**, this structure is populated by the library to return messages from the current session history. In this case, additional members such as the message identifier, timestamp, and token counts may contain

values assigned by the library. Applications should treat values returned by the library as read-only unless otherwise documented.

The *lpszContent* member specifies the text associated with the message. For the ANSI version of this structure, this string must be UTF-8 encoded. In the Unicode version, the string is specified as UTF-16 using a wide character pointer.

The *nContentLength* member specifies the content length in characters rather than bytes. If this value is zero, the library may determine the length from the null-terminated string. Applications should not assume that token counts are identical to character counts. The *nInputTokens* member, when available, reflects the tokenized size of the message content as determined by the model or provider.

The *dwReasoningLevel* member controls the reasoning effort requested for the message. Higher reasoning levels may improve the quality, accuracy, or depth of responses, particularly when using reasoning-capable models, but they can also increase response time and token usage costs. Lower reasoning levels generally favor faster responses and reduced usage costs. Not all providers or models support configurable reasoning levels, and unsupported values may be ignored by the provider. If this member is set to `LLM_REASONING_DEFAULT`, the library will use the default reasoning behavior for the current provider or model.

The *tTimestamp* member is a standard 64-bit Unix time value and can be used interchangeably with functions in the C standard library such as **gmtime** or **localtime**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps can be converted into a local date and time using the **ConvertTimestamp** method.

Unless otherwise noted, any reserved members should be set to zero before calling a method which accepts this structure. Applications should also avoid modifying structure members that are returned by the library unless the documentation for a specific method indicates that doing so is supported.

The *lpszContent* member of this structure is managed by the library. Applications should treat this value as read-only and must not attempt to modify the contents of the string. If the application needs to retain or modify this data, it should create its own copy.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SendMessageEx GetMessage, GetMessageById, GetFirstMessage, GetNextMessage, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_MODEL_INFO

The **LLM_MODEL_INFO** structure contains descriptive information and capability data for a language model.

```
typedef struct _LLM_MODEL_INFO
{
    DWORD    dwSize;
    DWORD    dwProviderId;
    DWORD    dwFlags;
    DWORD    dwStatus;
    DWORD    dwReserved;
    LONG     nMaxTokens;
    LONG     nContextSize;
    FLOAT    fTemperature;
    FLOAT    fMaxTemperature;
    FILETIME ftCreated;
    TCHAR    szName[LLM_MODEL_NAMELEN];
    TCHAR    szDisplayName[LLM_MODEL_NAMELEN];
    TCHAR    szOwner[LLM_MODEL_NAMELEN];
    TCHAR    szVersion[LLM_MODEL_VERLEN];
    TCHAR    szCreated[LLM_MODEL_DATELEN];
} LLM_MODEL_INFO, *LPLLM_MODEL_INFO;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_MODEL_INFO)` before the structure is used.

dwProviderId

The provider associated with the model. This member should be one of the LLM_PROVIDER values.

dwFlags

One or more capability flags associated with the model. This member may contain a combination of LLM_MODEL_FLAG values.

dwStatus

One or more status flags associated with the model. This member may contain a combination of LLM_MODEL_STATUS values.

dwReserved

A reserved value for future use. This member should always be zero.

nMaxTokens

The maximum number of tokens the model can generate in response to a request. A value of zero indicates that this limit is not specified.

nContextSize

The maximum context window size for the model, in tokens. If this value is zero, the context size is unspecified or the provider does not distinguish it from ***nMaxTokens***.

fTemperature

The default temperature value associated with the model. A value of zero may indicate that the value is not specified.

fMaxTemperature

The maximum supported temperature value for the model. A value of zero indicates that this limit is not specified.

ftCreated

The creation date and time associated with the model, expressed as a FILETIME value. If this member is zero, the creation time is not available.

szName

A null-terminated string which specifies the model identifier used by the provider, such as `gpt-5` or `claude-4-sonnet`.

szDisplayName

A null-terminated string which specifies an optional display name for the model. This value is intended for presentation to the user and may be an empty string.

szOwner

A null-terminated string which specifies the optional owner, vendor, or organization associated with the model. This value may be an empty string.

szVersion

A null-terminated string which specifies the optional version identifier associated with the model. If this information is not available, the value will be an empty string.

szCreated

A null-terminated string which specifies the optional model creation date in text form. If this information is not available, the value will be an empty string.

Remarks

This structure is used to return information about a model that is available through a specific provider. It is typically populated by the library when enumerating models or retrieving detailed information about a specific model. Applications should treat the contents of this structure as descriptive metadata and should not assume that every member will contain a meaningful value for every model.

The ***dwProviderId*** member identifies which service or platform supplies the model, using one of the `LLM_PROVIDER` constants. The ***dwFlags*** member describes the capabilities supported by the model, such as conversational input, structured output, reasoning, vision, tool calling, or audio support. The ***dwStatus*** member indicates the current state of the model, such as whether it is generally available, in preview, or deprecated. These values are defined by the library using the `LLM_MODEL_FLAG` and `LLM_MODEL_STATUS` constants.

The ***nMaxTokens*** value represents the maximum output token limit reported for the model. This limit only applies to generated content and may be lower than the model's total context window. The client can further restrict the length of generated responses if necessary. If this value is zero, it means the provider does not publish information about the maximum number of output tokens supported for a response. Internally, providers often impose generation limits that are lower than the total context size for a request, even if those limits are not publicly documented.

The ***nContextSize*** value represents the maximum number of tokens the model can process in a single request, including the system prompt, conversation history, user messages and generated output. Most cloud-based service providers offer models which support very large context windows while limiting the maximum number of tokens that can be generated in a single response. For example, a model may support a context window of 1,000,000 tokens, allowing it to process very large conversations or documents, while limiting individual responses to 128,000 output tokens.

The context window and maximum token values are advisory. They are used when the service

provider does not return detailed limit information for a model. These values should be treated as general guidance and are not enforced by the library. The actual limits may vary depending on the provider, model version, deployment, enabled features and account limits. A value of zero for either ***nMaxTokens*** or ***nContextSize*** means those limits cannot be determined for the model. For locally hosted models or cloud-based aggregators, these limits may also be configurable for a specific model.

The metadata strings are provided for identification and display purposes. The ***szName*** member contains the canonical model name used in API requests, while ***szDisplayName*** provides a more user-friendly label when one is available. Other fields, such as ***szOwner***, ***szVersion***, ***szCreated***, and ***ftCreated***, are optional and may not be supplied by every provider.

Applications should set the ***dwSize*** member before passing this structure to a method. Unless otherwise documented, reserved members should be set to zero. When this structure is returned by the library, string members are stored directly in the fixed-length buffers contained within the structure, and may be read safely by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumModels, GetModelInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_PROVIDER_INFO

The **LLM_PROVIDER_INFO** structure provides descriptive information and default connection settings for a language model provider.

```
typedef struct _LLM_PROVIDER_INFO
{
    DWORD   dwSize;
    DWORD   dwProviderId;
    DWORD   dwFlags;
    DWORD   dwReserved;
    LONG    nProtocol;
    LONG    nHostPort;
    TCHAR   szName[LLM_PROVIDER_NAMELEN];
    TCHAR   szHostName[LLM_PROVIDER_HOSTLEN];
    TCHAR   szDefaultModel[LLM_MODEL_NAMELEN];
} LLM_PROVIDER_INFO, *LPLLM_PROVIDER_INFO;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_PROVIDER_INFO)` before the structure is used.

dwProviderId

The provider identifier. This member should be one of the `LLM_PROVIDER` values. Refer to the providers page for a list of supported providers and their default models.

dwFlags

One or more provider capability flags. This member may contain a combination of `LLM_PROVIDER_FLAG` values.

dwReserved

A reserved value for future use. This member should always be zero.

nProtocol

The protocol used by the provider endpoint. Most cloud-based service providers require secure connections and applications should use `LLM_PROTOCOL_HTTPS`. For locally hosted models running on the same system or over a local network, `LLM_PROTOCOL_HTTP` can be used to establish a standard, non-secure connection.

nHostPort

The network port number associated with the provider endpoint. A value of zero indicates that the provider-specific default port should be used.

szName

A null-terminated string which specifies a canonical provider name, such as `"OpenAI"`. This value may be used for identification or display in the user interface.

szHostName

A null-terminated string which specifies the default host name or IP address for the provider endpoint. This value may be an empty string for custom or application-defined providers.

szDefaultModel

A null-terminated string which specifies the default model name associated with the provider. This value may be an empty string if the provider requires the application to explicitly specify a model

name.

Remarks

This structure is used to describe a language model provider, including its identity, endpoint information, and basic capabilities. It is typically populated by the library when enumerating known providers or retrieving information about a specific provider.

The ***dwProviderId*** member identifies the provider using one of the `LLM_PROVIDER` constants, while the ***dwFlags*** member indicates provider-level requirements or supported features, such as whether the provider requires TLS, requires authentication, supports enumerating models, or supports returning detailed model information. The ***nProtocol*** member identifies the default protocol used to communicate with the provider endpoint, such as HTTP or HTTPS.

The string members provide descriptive metadata about the provider. The ***szName*** member contains the canonical provider name. The ***szHostName*** member specifies the default network hostname associated with the provider, and ***szDefaultModel*** specifies the default model that may be used if the application does not explicitly choose one.

Not all providers supply the same level of detail. Some values may be empty or unspecified, particularly for custom, local, or application-defined providers. Applications should treat the contents of this structure as descriptive metadata and should not assume that every member will contain a meaningful value in all cases.

Applications should set the ***dwSize*** member before passing this structure to a method. Unless otherwise documented, reserved members should be set to zero. Because all string values are stored directly in the fixed-length buffers contained within the structure, the application may safely read or copy them as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumProviders, GetProviderInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_RESPONSE

The **LLM_RESPONSE** structure represents the response returned by a language model.

```
typedef struct _LLM_RESPONSE
{
    DWORD   dwSize;
    DWORD   dwRoleId;
    DWORD   dwFlags;
    DWORD   dwMessageId;
    DWORD   dwStatus;
    DWORD   dwElapsed;
    DWORD   dwReserved;
    TIME_T  tTimestamp;
    LONG    nOutputTokens;
    LONG    nContentLength;
    LPCTSTR lpszResponseId;
    LPCTSTR lpszModelName;
    LPCTSTR lpszStatusText;
    LPCTSTR lpszContent;
} LLM_RESPONSE, *LPLLM_RESPONSE;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_RESPONSE)` before the structure is used.

dwRoleId

The role associated with the response. This member will typically be set to `LLM_ROLE_ASSISTANT`, but may vary depending on the type of response returned.

dwFlags

Response flags. This member is reserved for future use and should typically be set to `LLM_RESPONSE_FLAG_NONE`.

dwMessageId

The identifier of the originating request message. This value corresponds to the *dwMessageId* member of the **LLM_MESSAGE** structure that initiated the request.

dwStatus

The response status code. This member should be one of the `LLM_RESPONSE values`, indicating how the request completed.

dwElapsed

The elapsed time, in milliseconds, required to process the request and generate the response.

dwReserved

A reserved value for future use. This member should always be zero.

tTimestamp

A 64-bit integer timestamp associated with the response. This member specifies when the response was received and the value is represented as the number of seconds from January 1, 1970 UTC.

nOutputTokens

The number of tokens generated by the response. This value is provided when token usage information is available.

nContentLength

The length of the response content, in characters, not including the terminating null character.

lpszResponseId

A null-terminated string which specifies a provider-defined identifier for the response. This value may be NULL.

lpszModelName

A null-terminated string which specifies the name of the model used to generate the response. This value may be NULL.

lpszStatusText

A null-terminated string which specifies a provider-specific status or diagnostic message. This value may be NULL.

lpszContent

A null-terminated string which specifies the response text returned by the model.

Remarks

The LLM_RESPONSE structure represents the result of a request submitted to a language model. It is typically returned by methods such as **SendMessageEx** and provides both the generated response content and additional metadata describing how the request was processed.

This structure is logically paired with the **LLM_MESSAGE** structure. While the **LLM_MESSAGE** structure defines the input provided to the model, the **LLM_RESPONSE** structure contains the output generated by the model along with status and diagnostic information. The ***dwMessageId*** member may be used to associate the response with the originating request message.

The ***lpszContent*** member points to a null-terminated string buffer which contains the response from the model. The ***nContentLength*** member specifies how many characters are in the response, not including the terminating null character. Line feed and carriage return characters are converted to the standard Windows CRLF end-of-line sequence, ensuring consistent formatting across different providers and applications. This also helps ensure compatibility with Windows common controls and other user interface components.

The ANSI version of this structure uses UTF-8 encoded strings, while the Unicode version uses UTF-16 encoded strings. Response text returned by a provider may contain Unicode characters, including international text, typographical symbols, emoji, mathematical symbols, or other non-ASCII characters. Applications should ensure that any user interface controls, files, or other components used to display or process response text are capable of handling Unicode text correctly.

Applications which cannot process Unicode text in the response can provide a system prompt requesting that the model limit its output to ASCII characters. However, because model output is non-deterministic, applications should not assume that non-ASCII characters will never be returned.

The ***dwStatus*** member indicates how the request completed. A value of `LLM_RESPONSE_COMPLETE` indicates that the response was generated normally, while other values may indicate that the response was truncated due to token limits, cancelled, timed out, or failed due to an error. Applications should always check this value before processing the response content.

The ***dwElapsed*** member provides the total time required to process the request, which can be useful for performance monitoring or diagnostics. The ***nOutputTokens*** member, when available, indicates the number of tokens generated by the model, which may be useful for usage tracking or cost estimation.

The ***tTimestamp*** member is a standard 64-bit Unix time value which indicates when the response was

received from the service provider. It can be used interchangeably with functions in the C standard library such as **gmtime** or **localtime**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps can be converted into a local date and time using the **ConvertTimestamp** method.

The string members, such as *lpszContent*, *lpszResponseId*, *lpszModelName*, and *lpszStatusText*, are managed by the library. Applications should treat these pointers as read-only and must not attempt to modify the contents of the referenced strings. If an application needs to retain or modify this data, it should create its own copy.

Calling the **ClearHistory** or **ResetSession** methods will invalidate these string pointers. Applications must not reference these values after the session history has been cleared.

Unless otherwise documented, reserved members should be set to zero before calling a method which accepts this structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstMessage, GetMessage, GetMessageById, GetNextMessage, SendMessageEx, LLM_MESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_SESSION

The **LLM_SESSION** structure contains extended configuration information used by the **ConnectEx** method to establish a session with a language model provider.

```
typedef struct _LLM_SESSION
{
    DWORD dwSize;
    DWORD dwProviderId;
    DWORD dwFlags;
    FLOAT fTemperature;
    FLOAT fTopP;
    LONG  nTopK;
    LONG  nTimeout;
    LONG  nIdleTime;
    LONG  nCacheTime;
    LONG  nMaxTokens;
    DWORD dwReasoningLevel;
    LPCTSTR lpszBaseUrl;
    LPCTSTR lpszModelName;
    LPCTSTR lpszApiKey;
    LPCTSTR lpszApiVersion;
    LPCTSTR lpszAppName;
    LPCTSTR lpszUserId;
    LPCTSTR lpszOrgId;
    LPCTSTR lpszProjectId;
    LPCTSTR lpszSystemPrompt;
} LLM_SESSION, *LPLLM_SESSION;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_SESSION)` before the structure is used. This value is validated by the library and is reserved for future structure versioning.

dwProviderId

The provider associated with the session. This member should be one of the `LLM_PROVIDER` values. If this value is `LLM_PROVIDER_NONE` the provider ID specified when calling the **LlmConnectEx** method will be used. Refer to the providers page for a list of supported providers and their default models.

dwFlags

Reserved flags for internal use. This member should be set to zero.

fTemperature

The temperature value used to control response randomness. A value of zero indicates that the provider or model default should be used. Unless you have a specific need to specify a temperature, it is recommended that you leave this value set to zero. Providers do not interpret temperature values identically and reasoning models may ignore the value entirely.

fTopP

The nucleus (top-p) sampling value. A value of zero disables top-p sampling and causes the default behavior to be used. It is recommended that you leave this value set to zero. Nucleus sampling is not used with most reasoning models and may be ignored. Although some providers still support setting a top-p value, most discourage using it with their current models.

nTopK

The top-k sampling value. A value of zero disables top-k sampling. It is recommended that you leave this value set to zero. Top-k sampling is not used with most reasoning models and in many cases the value is ignored. Although some providers still support setting a top-k value, most discourage using it with their current models.

nTimeout

The timeout period for requests, in seconds. A value of zero indicates that the default timeout should be used.

nIdleTime

The maximum idle time, in seconds, that an existing HTTP connection may be retained when the `LLM_OPTION_KEEPALIVE` connection option is enabled. If the elapsed idle time exceeds this value, the current connection will be closed and a new connection will be established for the next request. A value of zero indicates that the default idle timeout should be used.

nCacheTime

The length of time, in seconds, that the library may cache provider-specific information. This is primarily used when caching model lists returned by a provider. A value of zero indicates that the default cache behavior should be used.

nMaxTokens

The maximum number of tokens to request for generated output. A value of zero indicates that the provider or model default should be used.

dwReasoningLevel

Specifies the default reasoning level used for requests in this session. Higher levels may improve response quality for complex tasks but can increase latency and token usage.

lpszBaseUrl

A pointer to a null-terminated string which specifies the base URL for the provider endpoint. This member may be NULL to use the default endpoint for the selected provider.

lpszModelName

A pointer to a null-terminated string which specifies the model name to use for the session. For Microsoft Foundry models, this member typically specifies the deployment name. This member may be NULL if the provider defines a default model.

lpszApiKey

A pointer to a null-terminated string which specifies the API key or authentication token used to access the provider. This member may be NULL if authentication is not required or is provided by another mechanism.

lpszApiVersion

A pointer to a null-terminated string which specifies the API version associated with the provider. This member may be NULL if no explicit version is required.

lpszAppName

A pointer to a null-terminated string which specifies an optional application name associated with the session. When provided, this value may be included in the User-Agent string sent with requests. This value may be sanitized by the library before use and may be NULL.

lpszUserId

A pointer to a null-terminated string which specifies an optional user identifier associated with the session. This member may be NULL.

lpszOrgId

A pointer to a null-terminated string which specifies an optional organization identifier associated with the session. This member may be NULL.

lpszProjectId

A pointer to a null-terminated string which specifies an optional project identifier associated with the session. This member may be NULL.

lpszSystemPrompt

A pointer to a null-terminated string which specifies an optional system prompt used to define the initial behavior of the model for the session. This member may be NULL.

Remarks

This structure is used with the **ConnectEx** method to provide extended configuration information when creating a session. It allows an application to customize connection behavior, sampling parameters, model selection, authentication, and other optional settings without requiring a large number of method parameters. Applications that only need to specify the provider, base URL, model name, and API key may use the simpler **Connect** method instead.

The *dwProviderId* member should normally match the provider specified in the **ConnectEx** method call. A value of `LLM_PROVIDER_NONE` may be used when the application relies on the method parameter to select the provider. If conflicting provider values are specified, the method will fail.

In most cases, applications only need to initialize *dwSize*, *dwProviderId*, and the specific members they wish to override. All unused numeric members should be set to zero, and all unused string members should be set to NULL. The library will apply provider-specific defaults when optional values are not specified.

The *lpszApiKey* member is a string value that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries. To specify an environment variable, surround the name in percent (%) symbols, such as `"%OPENAI_API_KEY%"`. To specify a Windows Credential, surround it in braces, such as `"{MyAppName/OpenAI}".`

For production applications, it is recommended that you avoid using literal API key strings and instead use an environment variable or a value stored in the Windows Credential Manager. If you must provide the API key as a literal string, you should take additional steps to protect it, such as storing the key in an encrypted form and limiting access to authorized users or processes. Avoid embedding API keys directly in application source code, configuration files, installers or other content that could be exposed through version control systems, logs or public repositories.

The *dwReasoningLevel* member provides a provider-neutral hint that controls how much internal reasoning a model should use when generating a response. Higher levels may improve the quality of results for more complex tasks, but can also increase response time and token usage. Not all models support reasoning controls, and the exact behavior is provider-specific. In most cases, it is recommended that you set this value to LLM_REASONING_DEFAULT, allowing the provider or model to determine the appropriate level automatically.

The sampling-related members *fTemperature*, *fTopP*, and *nTopK* control how responses are generated. In most applications, adjusting the temperature alone is sufficient, while Top-P and Top-K are intended for more advanced tuning scenarios. If these values are not specified, the provider or model defaults are used. Not all providers or models support sampling parameters such as temperature, top-p or top-k. The library will only include those values when they are supported by the selected provider and model. For models that do not support these parameters, the session values are

ignored.

The connection-related members *lpszBaseUrl*, *nTimeout*, and *nIdleTime* control how requests are sent and how long an existing HTTP connection may be reused. When the `LLM_OPTION_KEEPALIVE` option is enabled, connections may be reused across multiple requests, subject to the idle timeout period.

The *nCacheTime* member controls how long the library may cache provider-specific information, such as model lists. This helps reduce unnecessary network requests when applications repeatedly query provider capabilities.

The *lpszAppName* member specifies the application name associated with requests sent by the control. This value is included as part of the default user-agent string and may be used for diagnostic and identification purposes by the provider. A recommended format is to specify both the application name and version number using a convention similar to standard user-agent strings, for example: `"MyApplication/1.0"`.

The *lpszUserId* member is a user identifier associated with requests sent to the current provider. Some providers allow applications to specify a user identifier as part of the request payload for diagnostics, analytics, rate limiting, or request tracking. Some providers may ignore the value or use it only for internal monitoring and logging purposes. Applications should avoid including sensitive or personally identifiable information in the user identifier value unless required by the provider.

The *lpszOrgId* member specifies an optional organization, tenant or account identifier used by some providers when processing requests. This is primarily intended for enterprise or multi-tenant environments where a provider account may be associated with multiple organizations or projects. Most applications will not need to set this value unless specifically required by the provider. If the current provider does not support organization identifiers, the value is ignored.

The *lpszProjectId* member specifies an optional project identifier used by some providers when processing requests. This is primarily intended for providers which support project-based authentication, billing or resource management. Most applications will not need to set this value unless specifically required by the provider or account configuration. If the current provider does not support project identifiers, the value is ignored.

The *lpszSystemPrompt* member may be used to define a session-level system prompt that establishes the behavior of the model. A system prompt provides instructions that are included with each request sent to the model. It can be used to define the expected role, tone, response style, formatting rules, or other application-specific guidance that should apply to the session. For example, an application might use a system prompt to request short responses, avoid Markdown formatting, return structured text, or answer only within the context of a specific feature or workflow.

The string members in this structure are supplied by the application and are treated as input values when the session is created. Applications should ensure that any referenced strings remain valid until the connection is established. The library makes an internal copy of these values, so the application does not need to maintain them after the connection completes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectEx, CreateSession

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_TOKEN_USAGE

The **LLM_TOKEN_USAGE** structure contains information about token usage for a client session.

```
typedef struct _LLM_TOKEN_USAGE
{
    DWORD   dwSize;
    DWORD   dwProviderId;
    DWORD   dwReserved;
    LONG    nMessageCount;
    LONG    nInputTokens;
    LONG    nOutputTokens;
    LONG    nContextUsed;
    LONG    nContextSize;
    TCHAR   szModelName[LLM_MODEL_NAMELEN];
} LLM_TOKEN_USAGE, *LPLLM_TOKEN_USAGE;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_TOKEN_USAGE)` before the structure is used.

dwProviderId

The provider identifier. This member should be one of the `LLM_PROVIDER` values. Refer to the providers page for a list of supported providers and their default models.

dwReserved

A reserved value for future use. This member will always be zero.

nMessageCount

The number of messages currently stored in the session history.

nInputTokens

The number of input tokens associated with system and user messages in the current session.

nOutputTokens

The number of output tokens generated in model responses for the current session.

nContextUsed

The total number of tokens currently used in the active chat context, including input messages and message history.

nContextSize

The maximum number of tokens supported by the current model for the active context. A value of zero indicates that this limit is not available.

szModelName

A null-terminated string which specifies the name of the model associated with the current session. This value may be empty if no model name is available.

Remarks

This structure is used to return token usage information for an active client session. It provides a summary of how many messages are currently in the session history, how many tokens have been used for input and output, and how much of the available context window is currently in use.

The ***nInputTokens*** member reflects tokens associated with user and system input, while

nOutputTokens reflects tokens generated by the model in its responses. The *nContextUsed* member represents the total token usage currently contributing to the active chat context, including prior message history that may be sent with each request.

The *nContextSize* member indicates the maximum context size supported by the current model, when that information is available. Applications may compare *nContextUsed* with *nContextSize* to estimate how much of the available context window has been consumed.

This structure is intended for informational and diagnostic use. Token counts are provider-dependent and may represent estimates rather than exact values, depending on the model or service in use. Applications should not rely on these values to enforce strict limits, as the provider ultimately determines whether a request exceeds its supported context or token constraints.

Applications should set the *dwSize* member before passing this structure to a method. Unless otherwise documented, reserved members should be set to zero. String values are stored directly in the fixed-length buffers contained within the structure and may be read safely by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SendMessage, SendMessageEx, GetTokenUsage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CLlmClient Class Constants

Constant	Value	Description
Ports		
LLM_PORT_DEFAULT	0	Use the provider-specific default port.
LLM_PORT_SECURE	443	Default port for secure HTTPS connections.
Protocols		
LLM_PROTOCOL_DEFAULT	0	Use the provider-specific default protocol.
LLM_PROTOCOL_HTTP	1	Use the HTTP protocol.
LLM_PROTOCOL_HTTPS	2	Use the HTTPS protocol.
LLM_PROTOCOL_UNKNOWN	-1	Unknown or invalid protocol.
Options		
LLM_OPTION_NONE	0	No options specified.
LLM_OPTION_NOCACHE	0x000001	Disable provider and model caching for the session.
LLM_OPTION_KEEPALIVE	0x000002	Enable persistent (keep-alive) HTTP connections.
LLM_OPTION_PROXY	0x000008	Use a configured proxy server for the connection.
LLM_OPTION_NOUSERAGENT	0x000020	Do not send a User-Agent header with the request.
LLM_OPTION_SECURE	0x001000	Force use of a secure (HTTPS) connection.
LLM_OPTION_DEFAULT	LLM_OPTION_NONE	Default option set (no flags enabled).
Providers		
LLM_PROVIDER_NONE	0	No provider specified.
LLM_PROVIDER_LOCAL	1	Local inference provider (e.g., LM Studio or Ollama).
LLM_PROVIDER_OPENAI	2	OpenAI provider using the ChatGPT models.

LLM_PROVIDER_ANTHROPIC	3	Anthropic provider using the Claude models.
LLM_PROVIDER_GOOGLE	4	Google AI provider using the Gemini models.
LLM_PROVIDER_MICROSOFT	5	Microsoft Foundry provider using OpenAI-compatible models.
LLM_PROVIDER_MISTRAL	6	Mistral AI provider using the Mistral models.
LLM_PROVIDER_UNKNOWN	((DWORD)-1)	Unknown or unrecognized service provider.
LLM_PROVIDER_DEFAULT	LLM_PROVIDER_LOCAL	Default service provider.
Reasoning		
LLM_REASONING_NONE	0	No reasoning, even if it is available.
LLM_REASONING_DEFAULT	1	Provider or model default.
LLM_REASONING_LOW	2	Prefer faster responses; reduces usage costs.
LLM_REASONING_MEDIUM	3	Balanced reasoning.
LLM_REASONING_HIGH	4	Prefer quality responses; increases usage costs.
LLM_REASONING_MAXIMUM	5	Maximum reasoning quality.
Temperature		
LLM_TEMPERATURE_MINIMUM	0.0	Deterministic output with minimal randomness.
LLM_TEMPERATURE_CONSERVATIVE	0.3	Low variability, more predictable responses.
LLM_TEMPERATURE_BALANCED	0.7	Balanced creativity and coherence.
LLM_TEMPERATURE_DEFAULT	1.0	Standard default sampling behavior.
LLM_TEMPERATURE_CREATIVE	1.2	Increased creativity and variation.
LLM_TEMPERATURE_MAXIMUM	2.0	Maximum randomness with highly varied output.
Top-P Sampling		

LLM_TOP_P_DISABLED	0.0	Nucleus sampling disabled.
LLM_TOP_P_MINIMUM	0.1	Very restrictive token selection.
LLM_TOP_P_CONSERVATIVE	0.8	Conservative token selection.
LLM_TOP_P_BALANCED	0.9	Balanced nucleus sampling.
LLM_TOP_P_CREATIVE	0.95	More diverse token selection.
LLM_TOP_P_MAXIMUM	1.0	Full probability distribution with no filtering.
LLM_TOP_P_DEFAULT	LLM_TOP_P_DISABLED	Default disables nucleus sampling.
Top-K Sampling		
LLM_TOP_K_DISABLED	0	Top-K sampling disabled.
LLM_TOP_K_MINIMUM	0	Equivalent to disabled sampling.
LLM_TOP_K_CONSERVATIVE	10	Restrict to top 10 tokens.
LLM_TOP_K_BALANCED	40	Balanced token selection range.
LLM_TOP_K_CREATIVE	60	More diverse token selection.
LLM_TOP_K_MAXIMUM	100	Wide token selection range.
LLM_TOP_K_DEFAULT	LLM_TOP_K_DISABLED	Default disables Top-K sampling.
Frequency		
LLM_FREQUENCY_DISABLED	0.0	No frequency penalty applied.
LLM_FREQUENCY_MINIMUM	0.0	Minimum frequency penalty.
LLM_FREQUENCY_MAXIMUM	2.0	Maximum frequency penalty.
LLM_FREQUENCY_DEFAULT	LLM_FREQUENCY_DISABLED	Default disables frequency penalty.
Presence		
LLM_PRESENCE_DISABLED	0.0	No presence penalty

		applied.
LLM_PRESENCE_MINIMUM	0.0	Minimum presence penalty.
LLM_PRESENCE_MAXIMUM	2.0	Maximum presence penalty.
LLM_PRESENCE_DEFAULT	LLM_PRESENCE_DISABLED	Default disables presence penalty.
Provider Flags		
LLM_PROVIDER_FLAG_NONE	0	No provider requirements specified.
LLM_PROVIDER_FLAG_REQUIRE_TLS	0x0001	Requires a secure TLS (HTTPS) connection.
LLM_PROVIDER_FLAG_REQUIRE_AUTH	0x0002	Requires authentication (e.g., API key or token).
LLM_PROVIDER_FLAG_API_VERSION	0x0004	Requires an explicit API version parameter.
LLM_PROVIDER_FLAG_LOCAL	0x0008	Locally hosted provider (LM Studio, Ollama, etc.)
LLM_PROVIDER_FLAG_MODEL_LIST	0x0010	Supports enumerating available models.
LLM_PROVIDER_FLAG_MODEL_INFO	0x0020	Supports retrieving detailed model information.
Model Options		
LLM_MODEL_OPTION_DEFAULT	0x0000	default list of chat models for a provider.
LLM_MODEL_OPTION_PREVIEW	0x0001	Include preview or experimental models.
LLM_MODEL_OPTION_STREAMING	0x0002	Include models that support streaming responses.
LLM_MODEL_OPTION_STRUCTURED	0x0004	Include models that support structured outputs.
LLM_MODEL_OPTION_REASONING	0x0008	Include reasoning-oriented models.
LLM_MODEL_OPTION_VISION	0x0010	Include models that support image input.
LLM_MODEL_OPTION_AUDIO	0x0200	Include models that support audio output (text-to-speech).

LLM_MODEL_OPTION_IMAGE	0x0400	Include models that support image generation or input.
LLM_MODEL_OPTION_VIDEO	0x0800	Include models that support video generation.
LLM_MODEL_OPTION_TRANSCRIPTION	0x1000	Include models that support speech-to-text transcription.
LLM_MODEL_OPTION_DEPRECATED	0x8000	Include deprecated or retired models.
LLM_MODEL_OPTION_ALL_MODELS	0xFFFF	Include all available model types.
Model Flags		
LLM_MODEL_FLAG_NONE	0x0000	No model capabilities defined.
LLM_MODEL_FLAG_CHAT	0x0001	Supports conversational text generation.
LLM_MODEL_FLAG_SYSTEM_ROLE	0x0002	Accepts a system role message in the input.
LLM_MODEL_FLAG_STREAMING	0x0004	Supports streaming responses.
LLM_MODEL_FLAG_STRUCTURED	0x0008	Supports structured output (e.g., JSON or schema).
LLM_MODEL_FLAG_VISION	0x0010	Supports image inputs (multimodal).
LLM_MODEL_FLAG_TOOLS	0x0020	Supports tool or function calling.
LLM_MODEL_FLAG_REASONING	0x0040	Supports reasoning-oriented generation.
LLM_MODEL_FLAG_RESPONSES	0x0080	Supports the Responses API (unified endpoint).
LLM_MODEL_FLAG_FUNCTIONS	0x0100	Supports OpenAI-style function calling with schemas.
LLM_MODEL_FLAG_AUDIO	0x0200	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	0x0400	Supports image generation.
LLM_MODEL_FLAG_VIDEO	0x0800	Supports video generation.

LLM_MODEL_FLAG_TRANSCRIPTION	0x1000	Supports speech-to-text transcription.
LLM_MODEL_FLAG_INSTRUCTIONS	0x2000	Supports a dedicated instruction field outside message lists.
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	0x4000	Uses `max_completion_tokens` instead of `max_tokens` in requests.
LLM_MODEL_FLAG_INFERRED	0x80000	Capabilities are inferred rather than explicitly defined.
Model Status		
LLM_MODEL_STATUS_NONE	0x0000	No status assigned.
LLM_MODEL_STATUS_AVAILABLE	0x0001	Model is actively available.
LLM_MODEL_STATUS_PREVIEW	0x0002	Model is in preview or experimental stage.
LLM_MODEL_STATUS_DEPRECATED	0x0004	Model is deprecated or legacy.
Limits		
LLM_MAX_MESSAGE_LENGTH	65536	Maximum length of a single input message (64 KB)
LLM_MAX_RESPONSE_LENGTH	262144	Maximum expected response length (256 KB)
LLM_MAX_SYSTEM_PROMPT_LENGTH	32768	Maximum length of the system prompt (32 KB)
LLM_MAX_CHAT_HISTORY	1000	Maximum number of messages stored per session.
LLM_MAX_TOKENS	200000	Maximum token limit per request.
Roles		
LLM_ROLE_UNKNOWN	0	Unknown or invalid role.
LLM_ROLE_USER	1	User input message.
LLM_ROLE_ASSISTANT	2	Assistant response message.
LLM_ROLE_SYSTEM	3	System-level instruction message.

LLM_ROLE_FUNCTION	4	Function call message (legacy)
LLM_ROLE_TOOL	5	Tool-generated message.
LLM_ROLE_TOOLUSER	6	User input directed to a tool.
LLM_ROLE_DEFAULT	LLM_ROLE_USER	Default role is user input.
Responses		
LLM_RESPONSE_UNKNOWN	0	No stop reason specified or unrecognized response state.
LLM_RESPONSE_COMPLETE	1	Response completed normally.
LLM_RESPONSE_LIMITED	2	Stopped due to token or context limits.
LLM_RESPONSE_ABORTED	3	Request was cancelled or interrupted.
LLM_RESPONSE_ERROR	4	An error occurred during processing.
LLM_RESPONSE_TIMEOUT	5	Request timed out.
Session		
LLM_SESSION_FLAG_NONE	0	No session flags specified.
LLM_SESSION_FLAG_DEFAULT	LLM_SESSION_FLAG_NONE	Default session behavior.
Cache		
LLM_INVALID_CACHE_TIME	-1	Indicates an invalid or unspecified cache duration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Providers

One of the first decisions you will need to make when using the SocketTools **CLlmClient** class is choosing which service provider to use. The class is designed to support a variety of providers and models, and the provider type is specified using one of following provider constants. The current version supports the following providers:

Constant	Value	Default Model	Description
LLM_PROVIDER_LOCAL	1	None	Locally hosted models using servers like LM Studio or Ollama. This is ideal for development, testing, offline use, or avoiding usage costs. A wide range of open-source models are available; for best results, start with smaller instruction-tuned models such as **qwen2.5-7b-instruct**, **llama-3.2-3b-instruct**, or **mistral-7b-instruct**.
LLM_PROVIDER_OPENAI	2	**gpt-5.4-nano**	OpenAI's public API (ChatGPT) is a recommended starting point for most developers due to ease of use, strong general performance, and comprehensive documentation. For initial development and testing, lightweight models such as the GPT-5 mini series provide fast response times and lower usage costs.
LLM_PROVIDER_ANTHROPIC	3	**claude-haiku-4-5**	Anthropic's Claude models are well suited for structured reasoning, conversational clarity, and safety-focused applications. For initial use, the Haiku models offer lower cost and faster response times while maintaining strong performance.
LLM_PROVIDER_GOOGLE	4	**gemini-2.5-flash**	Google's Gemini models are a good choice for multimodal applications and integration with the Google AI ecosystem. The Flash models are recommended for getting started due to their lower latency and reduced token costs.
LLM_PROVIDER_MICROSOFT	5	None	Microsoft Foundry OpenAI-compatible models are typically used in enterprise environments requiring Azure integration and regional deployment control. This option requires provisioning a model deployment within Foundry and specifying the deployment name and API version when connecting.
LLM_PROVIDER_MISTRAL	6	**mistral-small-3.2**	Mistral AI models provide efficient and cost-effective options for general-purpose and

			reasoning tasks. Smaller models are recommended when getting started due to their performance and lower cost.

Although SocketTools provides a default model for most providers, the model names and availability will change over time. The providers which do not have a default model assigned to them will always require a specific model name when establishing the connection. Some providers, notably Microsoft Foundry and locally hosted services, will explicitly require you to provide a model name to establish a connection.

It is important to note that the class is not limited only to those default models listed above. Developers are encouraged to enumerate available models and explicitly select a model that meets their requirements. The class provides methods designed to return information about all of the compatible models supported by a provider. For more information refer to the EnumModels method.

Choosing a Model

Most language model providers offer multiple models with different capabilities, performance characteristics, and pricing. Rather than selecting a specific model name immediately, it is often more useful to choose a general category of model based on your application's needs. Common model categories include:

- **Lightweight**. Optimized for speed and low cost. These are ideal for simple tasks such as formatting, summarization, classification, or user interface interactions where responsiveness is important. These are the default models assigned to supported providers. In most cases, we recommend you start with a lightweight model until you determine your application could benefit from a more capable model type.
- **Balanced**. General purpose models that provide good performance across a wide range of tasks. These are typically the best default choice for most applications which need to balance between performance and capability. Examples are `gpt-5-mini`, `claude-sonnet-4-6`, `gemini-3.1` and `mistral-medium` models.
- **Reasoning**. Designed for more complex tasks such as multi-step problem solving, analysis, and code generation. These models may produce more detailed responses but typically use more tokens and have higher latency and increased costs. Examples are `gpt-5`, `claude-opus-4-7`, `gemini-3.1-pro` and `mistral-large` models.
- **Multimodal**. Support input or output beyond plain text, such as images, audio, or structured data. These are useful when working with media content or advanced integrations.

In most cases, developers should begin with a lightweight or balanced model and only move to more advanced models when additional capability is required. The exact model names and capabilities will vary between providers, but these categories provide a useful framework for selecting an appropriate model. Applications should also avoid relying on strict token limits or attempting to precisely manage context usage. Providers ultimately determine request limits, and applications should be prepared to handle responses that are truncated or rejected when limits are exceeded.

API Keys and Authentication

Most language model providers require an API key (or authentication token) to access their services. An API key is a unique identifier associated with your account that is included with each request to authenticate the client and track usage.

When using the **CLlmClient** class, the API key is typically provided when establishing a session using the **Connect** method. The class will include this key in requests to the selected provider using the

appropriate authentication mechanism.

Because API keys grant access to external services and may incur usage charges, they should be handled carefully and not exposed unnecessarily. The class supports several methods for providing API keys, allowing applications to balance convenience and security.

Providing an API Key

The class supports the following methods for supplying an API key:

- **Literal string**
  The API key can be provided directly as a string when creating a session. This is the simplest approach and is often used for testing or development, but it is not recommended for production applications because the key may be exposed in application binaries or source code.
- **Environment variable**
  The API key can be stored in an environment variable and retrieved at runtime. This approach avoids embedding the key directly in the application and is commonly used in development environments, scripts, or deployment pipelines.
- **Windows Credential Manager**
  The API key can be stored securely using the Windows Credential Manager and retrieved by the application at runtime. This is the recommended approach for Windows applications because it provides secure storage and helps prevent accidental exposure of sensitive credentials.

Best Practices

- Do not embed API keys directly in source code for production applications.
- Use secure storage mechanisms, such as the Windows Credential Manager, whenever possible.
- Limit access to API keys and avoid sharing them between applications or users.
- Review provider usage policies and configure usage limits or alerts where available.

Each provider has its own process for creating and managing API keys. Refer to the provider documentation for details on generating keys, rotating credentials, and managing access.

Application Design

When designing an application that uses a language model provider, developers generally have two approaches for managing API keys:

- **Application-managed API key**
  The developer provides an API key that is used by the application for all requests. This approach is simple to implement and provides a consistent user experience, but the developer is responsible for all usage and any associated costs.
- **User-provided API key (Bring Your Own Key)**
  The application allows the end user to supply their own API key, typically through a configuration dialog or settings interface. In this case, the user is responsible for usage and costs, and the application acts as a client using the user's credentials.

Each approach has trade-offs. Using an application-managed key simplifies the user experience but may require safeguards such as rate limiting or usage monitoring. Allowing users to provide their own key can reduce operational costs and provide greater flexibility, but requires additional user interface considerations and clear instructions for obtaining and configuring API access.

Applications intended for a broad audience often prefer a built-in API key for ease of use, while developer-focused tools frequently support user-provided keys for flexibility.

Provider Overview

The following sections provide more detailed information about the supported providers. One of the advantages of the **CLlmClient** class is that applications are not limited to a single language model service. Different providers may offer advantages related to pricing, performance, regional availability, deployment flexibility, or model capabilities.

Supporting multiple providers can also provide greater flexibility for end users. Some users or organizations may already have existing accounts, billing arrangements, or infrastructure associated with a specific provider such as OpenAI, Google, Microsoft Foundry, or a locally hosted inference service. Applications can allow users to select the provider and model which best meets their requirements without requiring significant changes to application logic.

LLM_PROVIDER_LOCAL

This provider type is used for locally hosted language model inference. This is particularly useful for development, testing, and experimentation without relying on a cloud-based provider. Local hosting allows developers to evaluate model behavior, test integrations, and work offline or within a private environment. This requires you to install and configure an application on your local system or network which will provide the inference service. For the Windows platform, we recommend one of the following applications, which are free to install and use:

LM Studio

LM Studio provides a full-featured desktop environment for running and testing language models locally. It includes model management, a built-in server, and support for OpenAI-compatible APIs, making it a strong choice for development and evaluation.

- LM Studio
- LM Studio Documentation

Ollama

Ollama is a lightweight local model runtime that makes it easy to download and run models on a local system. It provides a simple API interface and is well suited for basic testing and self-hosted deployments.

- Ollama
- Ollama Documentation

LLM_PROVIDER_OPENAI

The OpenAI provider connects to OpenAI's public API and is generally the recommended starting point for most developers. It offers a wide range of well-documented models, consistent API behavior, and strong general-purpose performance. Developers who are new to using large language models or want a reliable default should typically begin here.

To get started, developers create an API key, select a model name, and connect to the default OpenAI endpoint. The API is widely supported and serves as the baseline for compatibility with many other providers.

- OpenAI Platform
- OpenAI Documentation
- Manage API Keys

LLM_PROVIDER_ANTHROPIC

The Anthropic provider connects to Anthropic's Claude API, which focuses on conversational models designed for safety and structured reasoning. Developers must create an API key and select a supported Claude model. The API is similar in concept to OpenAI but uses its own endpoints and request formats.

- Anthropic
- Claude Documentation

LLM_PROVIDER_GOOGLE

The Google provider connects to Google's Gemini API, which provides access to multimodal models and integrates with the broader Google AI ecosystem. Developers typically obtain an API key through Google AI Studio and use it to authenticate requests to the Gemini service.

- Google AI
- Google Documentation

LLM_PROVIDER_MICROSOFT

The Microsoft provider connects to a Microsoft Foundary OpenAI-compatible deployment. This option is often used in enterprise environments where Azure integration, regional deployment, or Microsoft-managed infrastructure is required. Unlike the public OpenAI API, Microsoft requires a resource-specific endpoint, a deployment name instead of a direct model name, and an explicit API version. Authentication is typically performed using an API key.

- Microsoft Foundry
- Azure OpenAI Reference

LLM_PROVIDER_MISTRAL

The Mistral provider connects to Mistral AI's hosted API service. Mistral models are designed to provide strong general-purpose and reasoning performance while maintaining efficient resource usage and competitive pricing. Developers who want a balance between capability, response speed, and lower operational cost may find these models to be a good alternative to larger commercial providers. Some organizations may prefer Mistral AI due to regional deployment considerations or compatibility with European-based infrastructure and services.

- Mistral AI
- Mistral Documentation
- Mistral Console

See Also

- Methods
- Constants
- Data Structures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Message History File Format

The **CLlmClient** class uses a JSON-formatted message history file to support exporting and importing conversation state between application sessions. This format is used by the **ExportHistory** and **ImportHistory** methods and is intended to provide a simple, human-readable representation of the conversation history associated with a client session.

A message history file contains both the messages sent by the client application and the responses generated by the model. Together, these entries represent the conversational context for the session. The file does not contain information about the client session, aside from the provider type and model name. It does not contain user names, authorization tokens or any information related to the service account.

The history file may contain sensitive or private information, including user prompts, model responses or application-specific data. Applications should take reasonable precautions to protect these files, particularly if they are stored, transmitted or made available for users to download.

The following example shows a simplified message history file containing a single message turn:

```
{
  "version": 1,
  "provider": 1,
  "created": 1715000000,
  "messages": [
    {
      "message": {
        "role": 1,
        "reasoning": 0,
        "flags": 0,
        "id": 1,
        "timestamp": 1715000001,
        "maxTokens": 0,
        "inputTokens": 12,
        "content": "Hello"
      },
      "response": {
        "role": 2,
        "flags": 0,
        "messageId": 1,
        "status": 1,
        "elapsed": 812,
        "timestamp": 1715000002,
        "outputTokens": 24,
        "model": "gpt-5.4-nano",
        "content": "Hi there."
      }
    }
  ],
  "messageCount": 1,
  "model": "gpt-5.4-nano"
}
```

File Structure

A message history file consists of a root JSON object which contains session metadata and an array of message turns.

version

The format version number used by the exported history file.

provider

The numeric provider identifier associated with the session.

created

A Unix timestamp value which indicates when the history file was created.

messages

An array containing the exported message turns.

messageCount

The number of message turns stored in the history file.

model

The default model name associated with the exported session.

Message Turns

Each entry in the **`messages`** array represents a complete message turn consisting of two objects:

- A **`message`** object which contains the client input
- A **`response`** object which contains the model output

The fields within these objects closely correspond to members of the **`LLM_MESSAGE`** and **`LLM_RESPONSE`** structures used internally by the API.

Applications should not depend on undocumented fields or the exact formatting of exported history files. Future versions of the library may add additional fields or metadata while maintaining compatibility with previous versions where possible.

Remarks

The message history format is intended primarily for exporting, importing and persisting conversation state between application sessions. Although the file format is human-readable JSON text, applications should avoid modifying exported history files unless they fully understand the relationship between the stored values.

Changing values such as the provider identifier, model name, message role or status values may produce undefined or unexpected results when the history is imported. Applications should not modify role identifiers or assume that numeric values used internally by the API will remain unchanged in future versions.

Modifying the message or response content may also invalidate related metadata such as token usage values. For example, changing the **`content`** field may result in incorrect **`inputTokens`** or **`outputTokens`** values being associated with the imported conversation history.

See Also

ExportHistory, ImportHistory, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Mail Message Class Library

Compose and parse standard MIME formatted email messages.

Reference

- Class Methods
- Error Codes

Library Information

Class Name	CMailMessage
File Name	CSMSGV12.DLL
Version	12.0.3024.1286
LibID	123A728A-D6CD-435F-A76D-0BA3C07DDFC1
Import Library	CSMSGV12.LIB
Dependencies	None
Standards	RFC 822, RFC 2045, RFC 2046, RFC 2047, RFC 2048

Overview

The Mail Message class provides an interface for composing and processing email messages and newsgroup articles which are structured according to the Multipurpose Internet Mail Extensions (MIME) standard. Using this class, an application can easily create complex messages which include multiple alternative content types, such as plain text and styled HTML text, file attachments and customized headers.

It is not required that the developer understand the complex MIME standard; a single method call can be used to create multipart message, complete with a styled HTML text body and support for international character sets. The Mail Message class can be easily integrated with the other mail related protocol libraries, making it extremely easy to create and process MIME formatted messages.

SocketTools also includes a class for managing a local message storage file that can be used to store and retrieve multiple messages. Methods are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values. For more information, refer to the CMessageStore class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Mail Message Class Methods

Class	Description
CMailMessage	Constructor which initializes the current instance of the class
~CMailMessage	Destructor which releases resources allocated by the class
Method	Description
AddHeaders	Add one or more header values to the current message
AppendText	Append text to the body of the current message part
AttachData	Attach the contents of a buffer to the current message
AttachFile	Attach a file to the current message
AttachHandle	Attach the current message handle to this instance of the class
AttachImage	Attach an inline image to the current message
CheckAuthenticationResults	Check the authentication status for the current message
Clear	Clear the current message, deleting all message parts
ClearText	Clear the body of the current message part
Close	Close the current message
CompareType	Compare the message contents against the specified content type
CompareText	Compare text in the body of the current message part
Compose	Compose a new message using the specified parameters
Copy	Copy the contents of another message into the current message
CreatePart	Create a new message part for the current message
DecodeText	Decode a base64 or quoted-printable encoded string
DeleteHeader	Delete the specified header field from the message
DeletePart	Delete the current message part
DetachHandle	Detach the handle for the current instance of this class
EncodeText	Encode a string using base64 or quoted-printable encoding
EnumAttachments	Enumerate all file attachments in the current message
EnumHeaders	Enumerate all header fields in the current message part
EnumRecipients	Enumerate addresses of all message recipients
Export	Export the current message to a string buffer or text file
ExtractAllFiles	Extract all file attachments in the message and store them in the specified directory
ExtractFile	Extract the file attachment from the current message part
FindAttachment	Search for a file attachment in the current message
FindText	Search for a regular expression or string literal in the current message text

FormatDate	Return a standard RFC 822 formatted date string
GetAllHeaders	Return the complete RFC 822 header values in a string buffer
GetAllRecipients	Return a comma-separated list of recipient addresses in a string buffer
GetAttachedFileName	Return the name of the file attachment for the current part
GetAuthenticationResults	Return information about the authentication status of the message
GetBoundary	Return the multipart message boundary string
GetContentDigest	Return encoded digest of message's content
GetContentLength	Return the length of the current message part content
GetContentType	Return the content type identifier for the current message part
GetDate	Return the date and time from the message header
GetErrorString	Return a description for the specified error code
GetExportOptions	Return a bitmask that describes current message export options
GetFileType	Return the content type for a specified file
GetFilteredText	Return the filtered contents for the current message part
GetFirstHeader	Return the first header field and value in the current message part
GetHandle	Return the message handle used by this instance of the class
GetHeader	Return the value of a specified header from the message
GetLastError	Return the last error code
GetLinks	Returns the hyperlinks in the current message part
GetNextHeader	Return the next header field and value in the current message part
GetPart	Return the current message part index
GetPartCount	Return the total number of message parts
GetSender	Return the email address of the message sender
GetSize	Return the size of the complete message in bytes
GetText	Return the text of the current message part
GetTextLength	Return the length of the text in the current message part
Import	Import a message from the specified text file, clipboard or buffer
IsContentType	Determine if the message part matches the specified content type
IsInitialized	Determine if the class has been successfully initialized
LocalizeText	Localize Unicode text to ANSI using a specific character set
NormalizeText	Normalize Unicode text, eliminating confusable homoglyphs and symbols
Open	Open and parse the specified text file as a MIME formatted message
ParseAddress	Parse the specified email address
ParseDate	Parse the specified RFC 822 formatted date string
ParseHeader	Parse the specified text and add to message header

ParseText	Parse the specified text and add to the current message
SetExportOptions	Specify a bitmask that describes current message export options
SetFileType	Set the content type for a specific file name extension
SetLastError	Set the last error code
SetDate	Set the current date in the header for the current message
SetHeader	Create or update a header field in the current message
SetPart	Set the current message part index for the current message
SetText	Create or update the current message body
ShowError	Display a message box with a description of the specified error

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AddHeaders Method

```
BOOL AddHeaders(
   LPCTSTR lpszHeaderList
);
```

The **AddHeaders** method adds one or more headers to the specified message.

Parameters

lpszHeaderList

A null-terminated string which specifies one or more header values which should be set for the specified message. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the string contains one or more invalid headers, the method returns a value of zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AddHeaders** method enables your application to set one or more header values by providing a list of name/value pairs separated by a colon. Multiple header values may be provided by separating them with a newline character. This method is similar to calling the **SetHeader** method for each value. When the list of header values is parsed, extraneous whitespace is ignored; however, if the header list contains invalid text (for example, a missing colon separating a header name from its value) the method will fail and an error will be returned.

This method will only add or update header values in the main header block for the message. It cannot be used to update header values in a specific section of a multipart message. If you need to add or change a header value in a specific part of the message, use the **SetHeader** method.

When adding custom, application-specific header values you should always prefix them with "X-" to avoid conflicting with standard headers.

Example

```
// Define a list of header values which should be included with the
// main header block for the message
LPCTSTR lpszHeaderList = _T("X-App-Sender: someuser@domain.tld\n") \
                         _T("X-App-Version: 1.5\n") \
                         _T("X-App-Mailer: AppMail (Win64)\n")

if (! pMessage->AddHeaders(lpszHeaderList))
{
    // Unable to add headers for this message
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHeaders, GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::CMailMessage Method

```
CMailMessage();
```

The **CMailMessage** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **MimeInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CMailMessage, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::~CMailMessage

```
~CMailMessage();
```

The **CMailMessage** destructor releases resources allocated by the current instance of the **CMailMessage** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CMailMessage** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. All handles that were client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CMailMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::AppendText Method

```
LONG AppendText(
   LPCTSTR lpszText
);
```

The **AppendText** method appends the specified text to the body of the current message part.

Parameters

lpszText

A null-terminated string which specifies the text to be appended to the current message part.

Return Value

If the method succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be appended to the current message part. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearText, CompareText, GetText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::AttachData Method

```
BOOL AttachData(
   LPBYTE lpBuffer,
   LONG cbBuffer,
   LPCTSTR lpszContentName,
   LPCTSTR lpszContentType,
   DWORD dwOptions
);

BOOL AttachData(
   CString& strBuffer,
   LPCTSTR lpszContentName,
   LPCTSTR lpszContentType,
   DWORD dwOptions
);
```

The **AttachData** method attaches the contents of the buffer to the message.

Parameters

lpBuffer

Pointer to a byte buffer which contains the data to be attached to the message. This parameter may be NULL, in which case no data is attached, but an additional empty message part will be created. An alternate form of this method accepts a **CString** object which contains the text to be attached to the message.

cbBuffer

An unsigned integer which specifies the number of bytes of data in the buffer pointed to by the *lpBuffer* parameter. If the *lpBuffer* parameter is NULL, this value must be zero.

lpszContentName

Pointer to a string which specifies a name for the data being attached to the message. This typically is used as a file name by the mail client to store the data in. If this parameter is NULL or an empty string then no name is defined and the data is attached as inline content. Note that if a file name is specified with a path, only the base name will be used.

lpszContentType

Pointer to a string which specifies the type of data being attached. The value must be a valid MIME content type. If this parameter is NULL or an empty string, then the buffer will be examined to determine what kind of data it contains. If there is only text characters, then the content type will be specified as "text/plain". If the buffer contains binary data, then the content type will be specified as "application/octet-stream", which is appropriate for any type of data.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_ATTACH_DEFAULT	The data encoding is based on the content type. Text data is not encoded, and binary data is encoded using the standard base64 encoding algorithm.
MIME_ATTACH_BASE64	The data is always encoded using the standard base64 algorithm, even if the buffer only contains printable text

	characters.
MIME_ATTACH_UUCODE	The data is always encoded using the uuencode algorithm, even if the buffer only contains printable text characters.
MIME_ATTACH_QUOTED	The data is always encoded using the quoted-printable algorithm. This encoding should only be used if the data contains 8-bit text characters.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **AttachData** method allows an application to attach data to the message as either a file attachment or as inline content. The recipient of the message will see the attached data in the same way that they would see a file attached to the message using the **AttachFile** method.

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, Export, ExtractFile, GetAttachedFileName, GetFileType, Import, SetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::AttachFile Method

```
BOOL AttachFile(
   LPCTSTR lpszFileName,
   DWORD dwOptions
);
```

The **AttachFile** method attaches the specified file to the message.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to be attached to the message.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_ATTACH_DEFAULT	The file attachment encoding is based on the file content type. Text files are not encoded, and binary files are encoded using the standard base64 encoding algorithm. This is the default option for file attachments.
MIME_ATTACH_BASE64	The file attachment is always encoded using the standard base64 algorithm, even if the attached file is a plain text file.
MIME_ATTACH_UUCODE	The file attachment is always encoded using the uuencode algorithm, even if the attached file is a plain text file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach data that is stored in a memory buffer rather than a file, use the **AttachData** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachData, Export, ExtractFile, GetAttachedFileName, GetFileType, Import, SetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::AttachHandle Method

```
VOID AttachHandle(
   HMESSAGE hMessage
);
```

The **AttachHandle** method attaches the specified message handle to the current instance of the class.

Parameters

hMessage

The handle to the message that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a message handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that message.

If a message handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous message from being destroyed, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hMessage*** parameter is presumed to be a valid message handle and no checks are performed to ensure that the handle is valid. Specifying an invalid message handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::AttachImage Method

```
BOOL AttachImage(
   LPCTSTR lpszImageFile,
   LPCTSTR lpszContentId
);
```

The **AttachImage** method attaches the specified file to the message as an inline image.

Parameters

lpszImageFile

Pointer to a string which specifies the name of the image file to be attached to the message. This parameter cannot be NULL and must specify the name of an existing file.

lpszContentId

An optional pointer to a null-terminated string which specifies the content ID associated with the inline image. If this parameter is NULL, a random content ID string will be automatically generated.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AttachImage** method enables an application to attach an inline image to the message. Unlike a normal file attachment, this method is designed to be used with HTML formatted email messages that display images attached to the message. The ***lpszContentId*** parameter specifies the content ID string that is used with the HTML image tag to reference that image.

If the specified message is not a multipart message, it is marked as multipart and the attached image file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach regular files to the message, use the **AttachFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachData, AttachFile, Compose

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::CheckAuthenticationResults Method

```
BOOL CheckAuthenticationResults(
    DWORD dwAuthFlags
);
```

The **CheckAuthenticationResults** method checks if the authentication status in the message matches one or more values.

Parameters

dwAuthFlags

A bitmask that specifies one or more authentication flags to check for. These can be any combination of the values listed below.

Value	Description
MIME_AUTH_SPF_PASSED	The message passed the Sender Policy Framework (SPF) check. The sender's IP address was authorized to send mail for the domain in the envelope sender address.
MIME_AUTH_SPF_FAILED	The message failed the SPF check. The sending mail server is not authorized by the domain's SPF record to send mail on its behalf.
MIME_AUTH_DKIM_PASSED	The message passed the DomainKeys Identified Mail (DKIM) check. The signature applied by the sender was successfully verified using the domain's public key.
MIME_AUTH_DKIM_FAILED	The message failed the DKIM check. The signature could not be verified or did not match the message content, suggesting it may have been altered.
MIME_AUTH_DMARC_PASSED	The message passed DMARC evaluation. The message aligned with a valid SPF or DKIM result based on the sender domain's policy.
MIME_AUTH_DMARC_FAILED	The message failed DMARC evaluation. It did not pass SPF or DKIM checks in a way that aligns with the domain's published policy.
MIME_AUTH_TEMP_ERROR	A temporary error occurred while evaluating one or more authentication mechanisms. This may indicate a DNS lookup timeout or other transient failure.
MIME_AUTH_PERM_ERROR	A permanent error occurred during authentication checks. This may indicate a malformed DNS record or invalid message structure that prevents proper evaluation.

Return Value

This method returns a non-zero value if one or more of the specified flags match the values found in the message's `Authentication-Results` header. If there is no match, the method returns zero. This method explicitly clears the last error code to assist in differentiating between an actual error condition (such as an invalid message handle) and a case where no authentication data matched.

Remarks

This method provides a convenient way to check if the authentication status of the message matches one or more values. In addition to the flags specified above, the following predefined flag combinations are also available:

- MIME_AUTH_PASSED is a combination of all passing results (SPF, DKIM, DMARC).
- MIME_AUTH_FAILED is a combination of all failing results.
- MIME_AUTH_ERROR is a combination of temporary or permanent errors.

This method allows for quick checks against common authentication outcomes using predefined flags. For example, to check whether the message failed authentication or encountered a temporary error, you could specify `MIME_AUTH_FAILED | MIME_AUTH_TEMP_ERROR`.

For more information about SPF, DKIM and DMARC, please refer to the **GetAuthenticationResults** method.

Example

```
if (mailMessage.Open(lpszFileName))
{
    // Check if any of the SPF, DKIM, or DMARC checks failed
    if (mailMessage.CheckAuthenticationResults(MIME_AUTH_FAILED))
        _tprintf(_T("This message has failed one or more authentication
checks\n"));

    mailMessage.Close();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

GetAuthenticationResults, GetHeader, Open

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::Clear Method

```
BOOL Clear();
```

The **Clear** method clears the header and body of the specified message, and deletes all message parts.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

Close, Export, Import, ParseText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ClearText Method

```
BOOL ClearText();
```

The **ClearText** method deletes the text from the body of the current message part.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AppendText, CompareText, GetText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::CompareType Method

```
BOOL CompareType(
   INT nMessagePart,
   LPCTSTR lpszContentType
);

BOOL CompareType(
   LPCTSTR lpszContentType
);
```

The **CompareType** method compares the content type for the message part against the specified content type.

Parameters

nMessagePart

An integer value which specifies which part of the message to compare against the content type. A value of zero returns compares the content type against the main message header, while a value greater than zero compares the content type against the content type for a specific part of a multipart message. If this parameter is omitted, the content type will be compared against the current message part.

lpszContentType

Pointer to a string which specifies the content type to compare against. This parameter cannot be NULL and it must specify a valid MIME media type. It is permitted to omit the media subtype, in which case the content type match will only be made against the base type. For example, both **"application/pdf"** and **"application"** are permitted as valid content types. Note that this is a prefix match, not a wildcard pattern match.

Return Value

The method will return a non-zero value if the specified content type matches the content type for the message part. If the content types do not match or one of the parameters is invalid, the method will return zero. To get extended error information, call the **GetLastError** method. If the parameters are valid but the content types do not match, the last error code will be set to NO_ERROR.

Remarks

The **CompareType** method is used to compare the content type for a specific part of a MIME-encoded message against a known media type. Multipurpose Internet Mail Extensions (MIME) defines a standard way of classifying the content of an email message or its individual parts, such as plain text, HTML, images, or file attachments.

A MIME media type, also referred to as a content type, is typically expressed in the form **type/subtype**. For example, the content type **text/plain** identifies unformatted plain text, while **text/html** identifies HTML formatted text. The content type for each message part is typically specified in its **Content-Type** header field. This method performs a case-insensitive comparison between the message part's declared content type and the type provided by the caller. It does not consider optional parameters, such as character encoding or boundary markers. For example, a content type of **"text/html; charset=UTF-8"** will match **"text/html"**.

The following table lists some of the most common MIME media types used in email messages:

Media Type	Description

`text/plain`	Plain text with no formatting
`text/html`	HTML-formatted text, commonly used for rich email content
`multipart/alternative`	Message that contains multiple representations of the same content, such as both plain text and HTML
`multipart/mixed`	Message that includes a combination of text and attachments
`multipart/related`	Message that includes related resources, such as inline images referenced by HTML content
`image/jpeg`	JPEG image
`image/png`	PNG image
`application/pdf`	PDF document
`application/octet-stream`	Generic binary data, typically used for file attachments

To obtain the full content type string for a message part, including any optional parameters, use the **GetHeader** method with the header name `"Content-Type"`.

Example

```
// Search the message for a PDF attachment by checking the content type
// for each part of a multipart message.
//
// If the content type matches "application/pdf" then that message part
// should contain a PDF attachment. We prompt the user where to store
// the file and then exit the loop.

INT nParts = mailMessage.GetPartCount();

for (INT nMessagePart = 1; nMessagePart < nParts; nMessagePart ++)
{
    if (mailMessage.SetPart(nMessagePart) == MIME_ERROR)
        break;

    if (mailMessage.CompareType(_T("application/pdf")))
    {
        TCHAR szAttachment[MAX_PATH] = _T("");

        // Get the name of the file attached to the message. This can be
        // useful for display purposes, but exercise caution if you want
        // to use that name to store the file locally.
        if (mailMessage.GetAttachedFileName(szAttachment, MAX_PATH))
        {
            _tprintf(_T("Found attachment \"%s\" in part %d\n"), szAttachment,
nMessagePart);

            // The recommended approach is to prompt your user for where
            // they would like you to store store the file attachment;
            // avoid using the remote filename directly without validating
            // or prompting the user.
            TCHAR szLocalFile[MAX_PATH] = _T("");

            OPENFILENAME ofn = {0};
            ofn.lStructSize = sizeof(OPENFILENAME);
            ofn.hwndOwner = NULL; // No owner window
```

```
            ofn.lpstrFilter = "Text Files (*.txt)\0*.txt\0All Files (*.*)\0*.*\0";
            ofn.lpstrFile = szLocalFile;
            ofn.nMaxFile = MAX_PATH;
            ofn.lpstrTitle = "Select File to Save";
            ofn.Flags = OFN_OVERWRITEPROMPT;

            // If the user cancels the dialog, we are finished
            if (!GetSaveFileName(&ofn))
                 break;

            // Extract the attachment to the file name specified by the user
            if (!mailMessage.ExtractFile(szLocalFile))
            {
                DWORD dwError = mailMessage.GetLastError();
                _tprintf(_T("Failed to extract file (Error 0x%08lX)\n"), dwError);
            }
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareText, ExtractFile, FindAttachment, GetAttachedFileName, GetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::CompareText Method

```
BOOL CompareText(
   LONG *lpnOffset,
   LPCTSTR lpszText,
   LONG cchText,
   BOOL bCaseSensitive
);
```

The **CompareText** method compares a text string against the contents of the current message part.

Parameters

lpnOffset

Pointer to a long integer which specifies the offset in the message at which to begin the comparison. This value will be updated when the method returns to indicate the offset position in the message where the comparison ended.

lpszBuffer

Pointer to a string buffer which contains the text that is to be compared against the body of the message.

cchBuffer

The number of characters in the buffer that should be compared against the body of the message.

bCaseSensitive

Boolean flag which specifies that the comparison should be case sensitive.

Return Value

If the text buffer matches the contents of the current message body, the method will return a non-zero value, and the ***lpnOffset*** argument will be set to position in the buffer where the match terminated. If the text buffer does not match, the method will return a value of zero, and the ***lpnOffset*** argument will be set to the position of the first non-matching character. The method will also return zero if one of the arguments is invalid. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AppendText, ClearText, GetText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::Compose Method

```
BOOL Compose(
   LPCTSTR lpszFrom,
   LPCTSTR lpszTo,
   LPCTSTR lpszCc,
   LPCTSTR lpszSubject,
   LPCTSTR lpszMessageText,
   LPCTSTR lpszMessageHTML,
   UINT nCharacterSet,
   UINT nEncodingType
);
```

The **Compose** method creates a new message using the specified parameters.

Parameters

lpszFrom

A null-terminated string which specifies the sender's email address. This parameter may be NULL, in which case no sender address will be included in the message header.

lpszTo

A null-terminated string which specifies one or more recipient addresses. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no recipient addresses will be included in the message header.

lpszCc

A null-terminated string which specifies one or more addresses that will receive a copy of the message in addition to the listed recipients. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no carbon-copy addresses will be included in the message header.

lpszSubject

A null-terminated string which specifies the subject of the message. This parameter may be NULL, in which case no subject will be included in the message.

lpszMessageText

A null-terminated string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this parameter is NULL or points to an empty string, then the message will have an empty body unless the ***lpszMessageHTML*** parameter is not NULL.

lpszMessageHTML

A null-terminated string which contains the message using HTML formatting. If the ***lpszMessageText*** parameter is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the ***lpszMessageText*** argument is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

nCharacterSet

A numeric identifier which specifies the character set to use when composing the message. A value of zero specifies that the default UTF-8 character set should be used. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* parameter cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* parameters.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly.

In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreatePart, Export, Import

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::Copy Method

```
BOOL Copy(
   CMailMessage *pMessage
);
```

The **Copy** method copies the contents of the specified message.

Parameters

pMessage

A pointer to a CMailMessage object which contains the message to be copied. If this argument is NULL, the current message contents will be cleared.

Return Value

If the message was successfully copied, the method will return a non-zero value. If the method fails, it will return value of zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to create a copy of the specified message, replacing the current message contents. Note that this method creates a duplicate of the message referenced by the *pMessage* object, and any subsequent changes to the contents of the original message will not be reflected in the copy of that message.

Example

```
CMailMessage *pMessage1 = new CMailMessage();
CMailMessage *pMessage2 = new CMailMessage();

// Compose a test message
pMessage1->Compose(_T("Bob Jones <bob@example.com>"),
                           _T("Tom Smith <tom@example.com>"),
                           NULL,
                           _T("This is a test message"),
                           _T("This is a test, this is only a test."));

// Create a copy of the message and change the subject
// The original subject in pMessage1 remains unchanged
if (pMessage2->Copy(pMessage1))
{
    CString &strSubject;
    if (pMessage2->GetHeader(0, _T("Subject"), strSubject))
    {
        strSubject = _T("Re: ") + strSubject;
        pMessage2->SetHeader(0, _T("Subject"), strSubject);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

Clear, Compose

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::CreatePart Method

```
INT CreatePart();

INT CreatePart(
   LPCTSTR lpszText,
   UINT nCharacterSet,
   UINT nEncodingType
);
```

The **CreatePart** method creates a new part for the specified message. If this the first part created for a message that does not have the multipart content type specified, the message is marked as multipart and the header fields are updated.

Parameters

lpszText

A null-terminated string which specifies the text to be included in the body of the new message part. If this parameter is NULL or points to an empty string, no text is added to the message part.

nCharacterSet

A numeric identifier which specifies the character set to use when composing the message. A value of zero specifies the character set should be the same character set used to initially compose the message. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data.

	This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the method succeeds, the return value is the new message part number. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, Compose, CreatePart, DeletePart, GetPart, GetPartCount, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::DecodeText Method

```
LONG DecodeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG cchOutput,
   UINT nCharacterSet,
   UINT nEncodingType
);

LONG DecodeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   CString& strOutput,
   UINT nCharacterSet,
   UINT nEncodingType
);
```

The **DecodeText** method decodes a string that was previously encoded using base64 or quoted-printable encoding.

Parameters

lpszInput

A null-terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be decoded. If this parameter is omitted or the value is -1, the entire length of the string up to the terminating null will be decoded.

lpszOutput

A null-terminated string buffer that will contain the decoded text. This buffer must be large enough to store all of the characters in the decoded text, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object instead of a fixed-length string buffer.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the decoded text and terminating null character. This value must be greater than zero. This parameter is not used if output buffer is specified as a **CString** object.

nCharacterSet

A numeric identifier which specifies the character set to use when decoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is decoded. If this value does not match the character set used when the text was originally encoded, the resulting output text may be invalid. This parameter may be omitted, in which case the method will default to using the UTF-8 character set.

nEncodingType

An integer value that specifies the encoding method used. This parameter may be omitted, or it may be one of the following values:

Value	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this method.

Return Value

If the input buffer can be successfully decoded, the return value is the length of the decoded output string. If the method returns zero, then no text was decoded and the output string buffer will be empty. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this method because the message parser will detect which character set and encoding was used, then automatically decode the message text if necessary.

The value of the ***nCharacterSet*** parameter does not affect the resulting output text, it is only used when decoding the input text. If the Unicode version of this method is called, the output text will be converted to UTF-16 and returned to the caller. If the ANSI version of this method is used, the decoded output will always be returned to the caller using the UTF-8 character set.

If the ***nCharacterSet*** parameter is specified as MIME_CHARSET_UTF16, the encoding type must be MIME_ENCODING_BASE64. Other encoding methods are not supported for Unicode strings and will cause the method to fail. In most cases, it is preferable to use MIME_ENCODING_BASE64 as the encoding method, with quoted-printable encoding only used for legacy support.

If an unsupported encoding type is specified, this method will return MIME_ERROR and the output text string will be empty. This method cannot be used to decode uuencoded text.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EncodeText, GetText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::DeleteHeader Method

```
BOOL DeleteHeader(
   LPCTSTR lpszHeader
);
```

The **DeleteHeader** method deletes the specified header field from the message.

Parameters

lpszHeader

Pointer to a string which specifies the header field that will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::DeletePart Method

```
BOOL DeletePart(
   INT nMessagePart
);
```

The **DeletePart** method deletes the specified message part from the multipart message. The memory allocated for the message part is released.

Parameters

nMessagePart

The message part index to delete.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method cannot be used to delete part zero, which is the main body of the message. Instead use the **Clear** method to clear the entire message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreatePart, CreatePart, GetPart, GetPartCount, Clear, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::DetachHandle Method

```
HMESSAGE DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message handle associated with the current instance of the class object. If there is no active client session, the value INVALID_MESSAGE will be returned.

Remarks

This method is used to detach a message handle created by the class for use with the SocketTools API. Once the message handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::EncodeText Method

```
LONG EncodeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG cchOutput,
   UINT nCharacterSet,
   UINT nEncodingType
);

LONG EncodeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   CString& strOutput,
   UINT nCharacterSet,
   UINT nEncodingType
);
```

The **EncodeText** method encodes a string using base64 or quoted-printable encoding.

Parameters

lpszInput

A null-terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be encoded. If this parameter is omitted or the value is -1, the entire length of the string up to the terminating null will be encoded.

lpszOutput

A null-terminated string buffer that will contain the encoded text. This buffer must be large enough to store all of the characters in the encoded text, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object instead of a fixed-length string buffer.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the encoded text and terminating null character. This value must be greater than zero.

nCharacterSet

A numeric identifier which specifies the character set to use when encoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is encoded.

nEncodingType

An integer value that specifies the encoding method used. It may be one of the following values:

Value	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set

	are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the input buffer can be successfully encoded, the return value is the length of the encoded output string. If the method returns zero, then no text was encoded and the output string buffer will be empty. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this method to encode text when using the **SetText** method. The class will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

If the *nCharacterSet* parameter parameter is non-zero, the method will encode the text using the specified character set. If the Unicode version of this method is called, the input text is converted to ANSI using the code page associated with the character set. If the ANSI version of this method is called, the input text is converted to Unicode using the system default code page, and then back to ANSI using the specified character set.

If the *nCharacterSet* parameter specifies the MIME_CHARSET_UTF16 character set, you must specify MIME_ENCODING_BASE64 as the encoding method. Other encoding methods are not supported for Unicode strings and will cause the method to fail. It is not recommended you encode text as UTF-16 unless there is a specific requirement to use that character set.

It is recommended that you use the MIME_CHARSET_UTF8 character set whenever possible. It is capable of encoding all Unicode code points, and is a standard for virtually all modern Internet applications. In most cases, it is preferable to use MIME_ENCODING_BASE64 as the encoding method, with quoted-printable encoding only used for legacy support.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DecodeText, GetText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::EnumAttachments Method

```
INT EnumAttachments(
   LPCTSTR* lpFileNames,
   INT nMaxFiles
);
```

The **EnumAttachments** method enumerates all of the file attachments in the current message.

Parameters

lpFileNames

A pointer to a an array of null-terminated strings that contain the names of the files attached to the message. If this parameter is NULL, the method will only return the number of files attached to the message.

nMaxFiles

An integer value that specifies the maximum size of the array of string pointers specified by the *lpFileNames* parameter. If this value is zero, the *lpFileNames* parameter is ignored and the method will only return the number of files attached to the message.

Return Value

If the method succeeds, the return value is the number of files attached to the message. If the message does not contain any file attachments, this method will return a value of zero. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Example

```
LPCTSTR lpszFiles[MAXFILES];

INT nFiles = pMessage->EnumAttachments(lpszFiles, MAXFILES);
if (nFiles == MIME_ERROR)
{
    DWORD dwError = pMessage->GetLastError();
    _tprintf(_T("Unable to enumerate attachments, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("There are %d files attached to the message\n"), nFiles);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, Export, ExtractAllFiles, GetAttachedFileName, Import

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::EnumHeaders Method

```
INT EnumHeaders();

INT EnumHeaders(
   LPCTSTR *lpHeaderList,
   INT nMaxHeaders
);
```

The **EnumHeaders** method returns a list of pointers to all header field names in the current message part. This can be used in conjunction with the **GetHeader** method to retrieve the values for every header in the message.

Parameters

lpHeaderList

Pointer to an array of pointers to null-terminated header field names. If this parameter is NULL, the method only returns the number of headers in the current message part.

nMaxHeaders

The maximum number of header fields which may be returned in the *lpHeaderList* parameter.

Return Value

If the method succeeds, the return value is the total number of headers that are defined in the current message part. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The values returned in the header list array must not be directly modified by the application. There is no specific order in which the header fields are enumerated by this method. The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

If this method is called without any arguments, it returns the number of headers in the current message part but does not return any data. This is useful for determining how much memory must be allocated for the *lpHeaderList* argument.

Example

```
// Determine the total number of headers in the current
// message part

nHeaders = pMessage->EnumHeaders();
if (nHeaders > 0)
{

    // Allocate memory for the list of headers

    lpHeaderList = (LPCTSTR *)malloc(nHeaders * sizeof(LPCTSTR));
    assert(lpHeaderList != NULL);

    // Retrieve the list of headers in the current
    // message part, and get their values

    pMessage->EnumHeaders(lpHeaderList, nHeaders);
```

```
    for (nIndex = 0; nIndex < nHeaders; nIndex++)
    {

         LPCTSTR lpszValue;
         lpszValue = pMessage->GetHeader(lpHeaderList[nIndex]);
         assert(lpszValue != NULL);

         printf("%s: %s\n", lpHeaderList[nIndex], lpszValue);

    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::EnumRecipients Method

```
INT EnumRecipients(
   LPCTSTR lpszExtraAddress,
   LPTSTR lpBuffer,
   LPDWORD lpcchBuffer
);

INT EnumRecipients(
   LPTSTR lpBuffer,
   LPDWORD lpcchBuffer
);

INT EnumRecipients(
   CStringArray& arrayRecipients,
   LPCTSTR lpszExtraAddress
);
```

The **EnumRecipients** method returns a null-terminated list of strings which contain the email address of each recipient for the specified message.

Parameters

lpszExtraAddress

A null-terminated string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpBuffer

Pointer to buffer which will contain zero or more null-terminated strings. The end of the string list is indicated by an additional terminating null. If this parameter is NULL, the method will calculate the minimum number of characters required to store the addresses and return the value in the *lpcchBuffer* parameter.

lpcchBuffer

A pointer to an unsigned integer which should be initialized to the maximum number of characters that can be copied into the buffer specified by the *lpBuffer* parameter. When the method returns, it will be updated to contain the actual number of characters copied into the buffer. If the *lpBuffer* parameter is NULL, then this value will contain the minimum number of characters required to store all of the recipient addresses in the current message.

arrayRecipients

A reference to a **CStringArray** object that will contain each of the recipient addresses specified in the message. This version of the method is only available if the program is compiled using Microsoft Foundation Classes (MFC).

Return Value

If the method succeeds, the return value is the total number of recipients for the current message. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnumRecipients** method returns a list of recipient email addresses for the specified message,

with each address being terminated by a null character. The end of the list is indicated by an additional null character. To determine the size of the buffer you should pass to this function, you can specify the *lpBuffer* parameter as NULL and initialize the value of the *lpcchBuffer* parameter to zero.

An alternative to the **EnumRecipients** method is the **GetAllRecipients** method that returns a comma-separated list of recipient addresses in a string buffer.

Example

```
LPTSTR lpRecipients = NULL;
DWORD cchRecipients = 0;
INT nRecipients = 0;

// Determine the number of characters that should be allocated to store
// all of the recipient addresses in the current message

nRecipients = pMessage->EnumRecipients(NULL, &cchRecipients);

// Allocate the memory for the string buffer that will contain all
// of the recipient addresses and call EnumRecipients
// again to store those addresses in the buffer

if (nRecipients > 0 && cchRecipients > 0)
{
    lpRecipients = (LPTSTR)::LocalAlloc(LPTR, (cchRecipients * sizeof(TCHAR));
    if (lpRecipients == NULL)
        return; // Virtual memory exhausted

    nRecipients = pMessage->EnumRecipients(lpRecipients,
                                           &cchRecipients);
}

// Move through the buffer, processing each recipient address
// that was returned

if (nRecipients > 0)
{
    LPTSTR lpszAddress = lpRecipients;
    INT cchAddress;

    while (lpszAddress != NULL)
    {
        if ((cchAddress = lstrlen(lpszAddress)) == 0)
            break;

        // lpszAddress specifies a recipient address
        // Advance to the next address string in the buffer
        lpszAddress += cchAddress + 1;
    }
}

if (lpRecipients != NULL)
{
    LocalFree((HLOCAL)lpRecipients);
    lpRecipients = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAllRecipients, GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::Export Method

```
BOOL Export(
   LPCTSTR lpszFileName,
   DWORD dwExportOptions
);

BOOL Export(
   HGLOBAL *lphBuffer,
   LPDWORD lpdwBufferSize,
   DWORD dwExportOptions
);

BOOL Export(
   LPTSTR lpszMessage,
   DWORD dwMessageSize,
   DWORD dwExportOptions
);

BOOL Export(
   CString& strMessage,
   DWORD dwExportOptions
);
```

The **Export** method exports the message to a file, the system clipboard or global memory buffer.

Parameters

lpszFileName

A null-terminated string which specifies the file name that will contain the message. If the file already exists, it will be overwritten with the message contents.

lpszMessage

A null-terminated string which will contain the message. An alternate form of this method accepts a pointer to a global memory handle which will contain the message data when the method returns.

dwMessageSize

An unsigned integer value which specifies the maximum size of the ***lpszMessage*** buffer.

dwExportOptions

An unsigned integer which specifies how the message will be exported. The following values may be combined using a bitwise Or operator:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.

MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the version of this method is used which exports the message to a pre-allocated string buffer, the *dwMessageSize* parameter must specify the maximum size of the buffer. If the method succeeds, the message will be copied to the buffer. If the method fails, the previous contents of the buffer will not be preserved. If the buffer provided is not large enough to store the entire message, the message contents will be truncated. The **GetSize** method can be used to determine the minimum size of the buffer required to store the complete message.

If the version of this method is used which returns an HGLOBAL memory handle to the caller, the handle must be dereferenced using the **GlobalLock** function. No changes should be made to this copy of the message. If you wish to modify the contents of the message, allocate a local buffer and make a copy of the message contents, or use the method which exports the message to a pre-allocated string buffer. Your application is responsible for calling **GlobalUnlock** and **GlobalFree** to unlock and free the handle when it is no longer needed.

Example

```
// Exports the message contents to a global buffer
HGLOBAL hgblMessage = NULL;
DWORD dwMessageSize = 0;

if (pMessage->Export(&hgblMessage, &dwMessageSize))
{
    LPBYTE lpMessage = (LPBYTE)GlobalLock(hgblMessage);

    if (lpMessage)
    {
        // Process the contents of the message
    }

    GlobalUnlock(hgblMessage);
    GlobalFree(hgblMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Import, GetSize, SetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ExtractAllFiles Method

```
INT ExtractAllFiles(
   LPCTSTR lpszDirectory
);
```

The **ExtractAllFiles** method extracts all of the file attachments in a message and stores them in the specified directory.

Parameters

lpszDirectory

A null-terminated string which specifies the name of the directory where the file attachments should be stored. If this parameter is NULL or points to an empty string, the attached files will be stored in the current working directory on the local system.

Return Value

If the method succeeds, the return value is the number of file attachments which were extracted from the message. If the message does not contain any file attachments, this method will return a value of zero. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will extract all of the files that are attached to the message and store them in the specified directory. The directory must exist and the current user must have the appropriate permissions to create files there. If a file with the same name as the attachment already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

To determine the file names for each of the attachments in a message, use the **EnumAttachments** method. To store a file attachment on the local system using a name that is different than the file name of the attachment, use the **ExtractFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, EnumAttachments, Export, ExtractFile, GetAttachedFileName, Import

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ExtractFile Method

```
BOOL ExtractFile(
   INT nMessagePart,
   LPCTSTR lpszFileName
);

BOOL ExtractFile(
   LPCTSTR lpszAttachment,
   LPCTSTR lpszFileName
);

BOOL ExtractFile(
   LPCTSTR lpszFileName
);
```

The **ExtractFile** method extracts a file attachment from the message and stores it on the local system.

Parameters

nMessagePart

An integer value that specifies the message part that contains the file attachment. This value may be -1, in which case the current message part will be used.

lpszAttachment

A null-terminated string that specifies the file name for the attachment in the message. If the file name of the attachment is not known, this parameter can be NULL or point to an empty string.

lpszFileName

A null-terminated string that specifies the name of a file on the local system. If this parameter is NULL or points to an empty string, the value of the ***lpszAttachment*** parameter will specify the name of the file in the current working directory. If both the ***lpszAttachment*** and ***lpszFileName*** parameters are NULL, the method will fail.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will store the contents of a file attachment in the current message part to the specified file on the local system. If a path is specified as part of the file name, it must exist and the current user must have the appropriate permissions to create the file. If a file with the same name already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this method will automatically decode the contents of the attachment.

If the ***nMessagePart*** parameter is specified, then an attachment in that message part will be stored in the specified file. If the message part does not contain a file attachment, the method will fail.

If the ***lpszAttachment*** parameter is specified, the method will search the entire message for an attachment with the same file name. The search is not case-sensitive, however it must match the attachment file name completely. This method will not match partial file names or names that include wildcard characters. If a match is found, the contents of that attachment will be stored in the file specified by the ***lpszFileName*** parameter.

To extract all of the files attached to a message in a single method call, use the **ExtractAllFiles**

method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, EnumAttachments, Export, ExtractAllFiles, Import

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::FindAttachment Method

```
INT FindAttachment(
   LPCTSTR lpszFileName
);
```

The **FindAttachment** method searches the message for an attachment with the specified file name.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file attachment to search for. This parameter should only specify a base file name; it should not include a file path and cannot be NULL.

Return Value

If the method succeeds, the return value is the message part number that contains an attachment that matches the specified file name. If the message does not contain an attachment with the specified file name, the method will return MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will search the message for a attachment that matches the specified file name. The search is not case-sensitive, however it must match the attachment file name completely. This method will not match partial file names or names that include wildcard characters. To obtain a list of all of the files attached to a message, use the **EnumAttachments** method.

Example

```
// The name of the file attachment to search for
LPCTSTR lpszFileName = _T("MyProject.docx");

// Search for the attached file and store it on the local system
INT nMessagePart = pMessage->FindAttachment(lpszFileName);
if (nMessagePart != MIME_ERROR)
{
    pMessage->SetPart(nMessagePart);

    if (pMessage->ExtractFile(lpszFileName) != MIME_ERROR)
        _tprintf(_T("Saved file attachment %s\n"), lpszFileName);
    else
    {
        _tprintf(_T("Unable to save file attachment %s\n"), lpszFileName);
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, ExtractAllFiles, ExtractFile, GetAttachedFileName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::FormatDate Method

```
LPTSTR FormatDate(
   LONGLONG nSeconds,
   LONG nTimezone,
   LPTSTR lpszDate,
   INT cchMaxDate
);

LPTSTR FormatDate(
   LONGLONG nSeconds,
   LPTSTR lpszDate,
   INT cchMaxDate
);

LPTSTR FormatDate(
   LONGLONG nSeconds,
   LONG nTimezone,
   CString& strDate
);

LPTSTR FormatDate(
   LONGLONG nSeconds,
   CString& strDate
);
```

The **FormatDate** converts the specified date, expressed as the number of seconds since 1 January 1970, into a string compatible with the RFC 822 standard format for email messages.

Parameters

nSeconds

A 64-bit integer which specifies the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C/C++ time methods. If the value of this parameter is zero, the current date and time is used.

nTimezone

A 32-bit integer which is set to the difference in seconds between the specified date's timezone and Coordinated Universal Time. A value of zero specifies Coordinated Universal Time, while a positive value specifies a timezone west of UTC, and a negative value specifies a timezone east of UTC. For example, the Eastern timezone would be specified as the value 18000 and the Pacific timezone would be the value 28800. If this argument is omitted from the method, then the current timezone is used.

lpszDate

A buffer which will contain the formatted date as a null-terminated string. This parameter cannot be a NULL pointer.

cchMaxDate

The maximum number of characters, including the terminating null character, which may be copied into the date string buffer.

Return Value

If the method succeeds, a pointer to the date string buffer is returned. If the method fails, a NULL pointer is returned. To get extended error information, call **GetLastError**.

Remarks

The date string is returned in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. This format is as follows:

```
www, dd mmm yyyy hh:mm:ss [-]zzzz
```

Each part of the date string is defined as follows:

Format	Description
www	Weekday
dd	Day
mmm	Month
yyyy	Year
hh	Hour (24-hour clock)
mm	Minutes
ss	Seconds
zzzz	Timezone

The weekday and month are displayed using standard three-character English abbreviations. The timezone is displayed as the difference (in hours and minutes) between the specified timezone and Coordinated Universal Time. For example, if the timezone is eight hours west of Coordinated Universal Time, the ***nTimezone*** value would be 28800. This would be displayed as -0800 in the formatted date string.

Note that the format of the date string is defined by the RFC 822 standard, and is not affected by localization settings on the host system.

Previous versions of this method represented date and time values as 32-bit integers, which limited the representable range to dates before 19 January 2038 (the so-called "Year 2038 problem"). After this date, 32-bit time values will overflow, resulting in incorrect date calculations and potential failures in software that processes email message dates. By contrast, this 64-bit version uses 64-bit integer values, matching the standard **`time_t`** type used by modern C/C++ libraries. This not only eliminates the Year 2038 limitation but also ensures compatibility with current and future systems, allowing correct handling of dates both far in the past and well into the future. Whenever possible, applications should transition to these 64-bit functions to ensure long-term reliability and portability.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ParseDate, GetDate, SetDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetAllHeaders Method

```
INT GetAllHeaders(
   LPTSTR lpszHeaders,
   INT nMaxLength
);

INT GetAllHeaders(
   CString& strHeaders
);
```

The **GetAllHeaders** method returns the complete RFC 822 header values in a string buffer.

Parameters

lpszHeaders

Pointer to string buffer which will contain the header values for the current message. This parameter may be NULL, in which case the method will calculate the number of characters needed to store the complete header block.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the ***lpszHeaders*** string. If the ***lpszHeaders*** parameter is NULL, this value must be zero. If the ***lpszHeaders*** parameter is not NULL, this value must be large enough to store the entire list of addresses.

strHeaders

A **CString** object that will contain the header values for the current message. This version of the method is available if MFC or ATL is being used by the application, and the memory required to store the complete header block will be calculated automatically.

Return Value

If the method succeeds and the ***lpszHeaders*** parameter is NULL, the return value is the minimum number of characters that should be allocated to store all of the header values, including the terminating null character. If the ***lpszHeaders*** parameter is not NULL, then the return value is the number of characters copied into the string, not including the terminating null character. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetAllHeaders** method will return all of the RFC 822 header values in a string buffer. This includes the message headers that are most commonly referred to, such as the To, From and Subject headers. Each header and its value are separated by a colon, and terminated with a carriage return and linefeed (CRLF) pair.

The headers and their values returned by this method will not be identical to the header block in the original message. If a header value is split across multiple lines, this method will fold the text, returning the complete header value on a single line of text and removing any extraneous whitespace. If the header value has been encoded by the mail client, this method will return the decoded value, not the original encoded value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetAllRecipients Method

```
INT GetAllRecipients(
   LPCTSTR lpszExtraAddress,
   LPTSTR lpszRecipients,
   INT nMaxLength
);

INT GetAllRecipients(
   CString& strRecipients,
   LPCTSTR lpszExtraAddress
);
```

The **GetAllRecipients** method returns a comma-separated list of recipient addresses in a string buffer.

Parameters

lpszExtraAddress

A null-terminated string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpszRecipients

Pointer to string buffer which will contain a comma-separated list of email addresses when the method returns. This parameter may be NULL, in which case the method will calculate the number of characters needed to store the complete list.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the *lpszRecipients* string. If the *lpszRecipients* parameter is NULL, this value must be zero. If the *lpszRecipients* parameter is not NULL, this value must be large enough to store the entire list of addresses.

Return Value

If the method succeeds and the *lpszRecipients* parameter is NULL, the return value is the minimum number of characters that should be allocated to store the list recipient addresses, including the terminating null character. If the *lpszRecipients* parameter is not NULL, then the return value is the number of characters copied into the buffer, not including the terminating null character. If the method returns a value of zero, then the specified message has no recipients. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetAllRecipients** method is useful for creating a list of message recipients that can be passed to methods like **CSmtpClient::SendMessage**. If you wish to dynamically allocate the string buffer that will contain the list of recipients, then the *lpszRecipients* parameter should be NULL and the *nMaxLength* parameter should have a value of zero. The method will then return the recommended size of the buffer that should be allocated. This value is guaranteed to be large enough to store the entire list of message recipients, including the terminating null character.

Example

```
CString strRecipients;
```

```
if (pMessage->GetAllRecipients(strRecipients) < 1)
{
    // There are no recipients for the current message
    AfxMessageBox(IDS_ERROR_NO_RECIPIENTS);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumRecipients, GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetAttachedFileName Method

```
BOOL GetAttachedFileName(
   LPTSTR lpszFileName,
   INT cchFileName
);

BOOL GetAttachedFileName(
   CString& strFileName
);
```

The **GetAttachedFileName** method returns the file name for the attachment to the current message part.

Parameters

lpszFileName

Pointer to a buffer that will contain the current attachment file name as a string. An alternate form of this method accepts a **CString** object which will contain the file name when the method returns.

cchFileName

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The method will first try to get the filename from the Content-Disposition header field. If this field does not exist, it then attempts to get the name from the Content-Type header field. If neither field exists, the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, GetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetAuthenticationResults Method

```
DWORD GetAuthenticationResults();
```

The **GetAuthenticationResults** method returns information about the authentication status of the message.

Parameters

None.

Return Value

This method returns one or more flags based on the value of the `Authentication-Results` header in the message. This method explicitly clears the last error code to assist in differentiating between an error condidition, such as an invalid message handle, and when no authentication information was available. The following flags are defined:

Value	Description
MIME_AUTH_NONE	The message does not contain an Authentication-Results header or does not include any recognizable authentication types and their associated results.
MIME_AUTH_SPF_PASSED	The message passed the Sender Policy Framework (SPF) check. The sender's IP address was authorized to send mail for the domain in the envelope sender address.
MIME_AUTH_SPF_FAILED	The message failed the SPF check. The sending mail server is not authorized by the domain's SPF record to send mail on its behalf.
MIME_AUTH_DKIM_PASSED	The message passed the DomainKeys Identified Mail (DKIM) check. The signature applied by the sender was successfully verified using the domain's public key.
MIME_AUTH_DKIM_FAILED	The message failed the DKIM check. The signature could not be verified or did not match the message content, suggesting it may have been altered.
MIME_AUTH_DMARC_PASSED	The message passed DMARC evaluation. The message aligned with a valid SPF or DKIM result based on the sender domain's policy.
MIME_AUTH_DMARC_FAILED	The message failed DMARC evaluation. It did not pass SPF or DKIM checks in a way that aligns with the domain's published policy.
MIME_AUTH_TEMP_ERROR	A temporary error occurred while evaluating one or more authentication mechanisms. This may indicate a DNS lookup timeout or other transient failure.
MIME_AUTH_PERM_ERROR	A permanent error occurred during authentication checks. This may indicate a malformed DNS record or invalid message structure that prevents proper evaluation.

Remarks

Modern email systems use a set of authentication mechanisms to help verify that a message was sent from a legitimate source and not forged by a spammer or attacker. The **GetAuthenticationResults** method enables you to easily determine which of these mechanisms have passed a check and which of them have failed. It evaluates the following authentication information reported by the mail server:

SPF
Sender Policy Framework allows the owner of a domain to specify which mail servers are authorized to send email on its behalf. If the sending server is not listed in the domain's SPF record, the message may be marked as suspicious. SPF helps prevent spoofing of the envelope sender address, which is often used in phishing attacks.

DKIM
DomainKeys Identified Mail uses public-key cryptography to sign specific parts of the email, such as headers and body content. The signature is added by the sending mail server, and receiving systems use DNS to retrieve the sender's public key to verify that the message has not been tampered with. A valid DKIM signature helps confirm that the message was truly sent by the claimed domain.

DMARC
Domain-based Message Authentication, Reporting, and Conformance builds on SPF and DKIM by defining a policy for how a domain wants email receivers to handle messages that fail authentication checks. It also ensures that the visible From: address matches the domain that passed SPF or DKIM (a concept called alignment). DMARC allows domain owners to monitor abuse and enforce stronger protection against spoofing.

Checking these authentication results can help identify spoofed or forged messages, improve filtering decisions, and increase overall mail security. While not every legitimate message will have all three, failure of all mechanisms, especially from domains that publish strict DMARC policies, should be treated as suspicious.

While these authentication methods are useful signals, failing one of them does not necessarily mean a message is malicious. Many legitimate messages, even from well-known companies, may have misconfigured SPF records, missing DKIM signatures, or inconsistent DMARC policies. Conversely, a message that passes all checks could still be unwanted or deceptive. These results should be treated as advisory signals rather than absolute indicators of legitimacy. Developers are encouraged to consider authentication status alongside other factors, such as message content, sender reputation, and user reporting.

To simplify checking authentication outcomes, the following predefined flag combinations are also available:

- MIME_AUTH_PASSED is a combination of all passing results (SPF, DKIM, DMARC).
- MIME_AUTH_FAILED is a combination of all failing results.
- MIME_AUTH_ERROR is a combination of temporary or permanent errors.

For a simplified boolean check using these flags, see the **CheckAuthenticationResults** method.

Example

```
if (mailMessage.Open(lpszFileName))
{
    // Get the authentication results for the message
    DWORD dwAuthFlags = mailMessage.GetAuthenticationResults();

    // Check if any of the SPF, DKIM, or DMARC checks failed
    if (dwAuthFlags & MIME_AUTH_FAILED)
```

```
        _tprintf(_T("This message has failed one or more authentication
checks\n"));

    mailMessage.Close();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CheckAuthenticationResults, GetHeader, Open

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetBoundary Method

```
LPCTSTR GetBoundary();
```

The **GetBoundary** method returns a pointer to the boundary string used to separate the parts of a multipart message.

Parameters

None.

Return Value

If the method succeeds, the return value is a pointer to the boundary string. If the method fails, a NULL pointer is returned. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreatePart, GetHeader, GetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetContentDigest Method

```
INT GetContentDigest(
   LPTSTR lpszDigest,
   INT cchDigest
);

INT GetContentDigest(
   CString& strDigest
);
```

The **GetContentDigest** method returns an encoded digest of the message.

Parameters

lpszDigest

Pointer to a string buffer to contain the MD5 digest for the specified message. An alternate form of this method accepts a **CString** object which will contain the digest when the method returns.

cchDigest

Maximum length of the digest string, in bytes.

Return Value

If the method succeeds, the return value is the length of the message digest string. A value of zero specifies that there is no MD5 digest for the current message. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns the value of the Content-MD5 header field in the main body of the message. If the header exists, it contains the MD5 digest for the message as defined in RFC 1864.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetContentLength Method

```
LONG GetContentLength();
```

The **GetContentLength** method returns the size of the current message part content in bytes.

Parameters

None.

Return Value

If the method succeeds, the return value is the content length. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This function will return the size of the content in the current message part as the number of bytes and does not account for any Unicode conversion of text. Exercise caution when using this function to determine the size of the buffer that should be allocated for a method like **GetText**. You should always allocate enough memory to accommodate any potential text conversion and decoding which may occur.

If you need to know the actual length of the decoded text in the current message part, use the **GetTextLength** method. That method will return the number of characters (not bytes) in the message after they have been decoded and the text has been converted to Unicode.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

GetHeader, GetPart, GetText, GetTextLength, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetContentType Method

```
INT GetContentType(
   INT nMessagePart,
   LPTSTR lpszSubType,
   INT cchSubType,
   LPTSTR lpszParams,
   INT cchParams
);

INT GetContentType(
   INT nMessagePart,
   LPTSTR lpszSubType,
   INT cchSubType,
);

INT GetContentType();
```

Parameters

nMessagePart

An integer value which specifies the message part. Message parts are zero-based with the part zero being the primary content and additional parts typically containing message content and file attachments. A value of MIME_CURRENT_PART (-1) specifies the content type for the current message part should be returned.

lpszSubType

A null-terminated string buffer that will receive the content subtype, such as "html" or "plain". This parameter may be NULL if the subtype is not required.

cchSubType

The maximum number of characters that can be copied into the ***lpszSubType*** buffer, including the terminating null character. If the buffer is too small, the method will return an error.

lpszParams

A null-terminated string buffer that will receive any parameters associated with the content type, such as `charset=utf-8`. This parameter may be NULL if parameters are not needed.

cchParams

The maximum number of characters that can be copied into the ***lpszParams*** buffer, including the terminating null character. If the buffer is too small, the method will return an error.

Return Value

The method returns an integer value that identifies the top-level media type for the current message part. This value may be one of the following predefined constants:

Value	Description
MIME_CONTENT_TEXT	Textual content (e.g.: text/plain, text/html)
MIME_CONTENT_IMAGE	Image content (e.g.: image/jpeg, image/png)
MIME_CONTENT_AUDIO	Audio content (e.g.: audio/mpeg)
MIME_CONTENT_VIDEO	Video content (e.g.: video/mp4)
MIME_CONTENT_APPLICATION	Application-specific data (e.g.: application/pdf)

MIME_CONTENT_MESSAGE	Embedded message (e.g.: message/rfc822)
MIME_CONTENT_MULTIPART	Multipart content (e.g.: multipart/alternative)
MIME_CONTENT_WIDETEXT	Extended text using wide character encoding
MIME_CONTENT_UNKNOWN	The content type is unknown or could not be determined

If the method fails, it returns MIME_ERROR. To retrieve extended error information, call **GetLastError**.

Remarks

The **GetContentType** method parses the `Content-Type` header of the current message part and returns the content type information in a normalized form. If requested, the subtype name and any optional parameters are copied to the caller-supplied buffers. This method eliminates the need for manual parsing of MIME headers and is useful for determining how message parts should be processed based on their media type. For example, consider a message part with:

```
Content-Type: text/html; charset=utf-8
```

This method will return `MIME_CONTENT_TEXT`, and optionally a subtype of `"html"`, and the parameter string `"charset=utf-8"`. If only the numeric media type is required for use in a `switch` or `if-then-else` statement, the content type and parameters buffers may be omitted or passed as `NULL`.

If you wish to determine if the content type for a message part matches a specific value, you can use the **IsContentType** method.

Example

```
TCHAR szSubType[32] = {0};
TCHAR szParams[64] = {0};

// Check if the current message part is plain text; szParams may
// contain the original character set used
INT nContentType = mailMessage.GetContentType(
                        MIME_CURRENT_PART,
                        szSubType, _countof(szSubType),
                        szParams, _countof(szParams));

if (nContentType == MIME_CONTENT_TEXT && lstrcmpiA(szSubType, "plain") == 0)
{
    // Get the size of the decoded/converted message text
    LONG cchMessage = mailMessage.GetText(NULL, 0);
    if (cchMessage < 1)
    {
        _tprintf(_T("Message part %d does not contain any text\n"), nMessagePart);
        return;
    }

    // Allocate a buffer for the message text on the heap
    LPTSTR pszMessage = (LPTSTR)malloc((cchMessage + 1) * sizeof(TCHAR));
    if (pszMessage == NULL) // Virtual memory exhausted
        return;

    // Copy the message contents to the buffer
    cchMessage = mailMessage.GetText(pszMessage, cchMessage + 1);
    if (cchMessage > 0)
    {
        // Write the contents of the message part to the console
```

```
        fwrite(pszMessage, sizeof(TCHAR), cchMessage, stdout);
        fflush(stdout);
    }

    // Free the memory allocated for our text buffer
    free(pszMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsContentType, GetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetDate Method

```
INT GetDate(
   LPTSTR lpszDate,
   INT nMaxLength,
   BOOL bLocalize
);

INT GetDate(
   CString &strDate,
   BOOL bLocalize
);
```

The **GetDate** method returns the current message date and time.

Parameters

lpszDate

A null-terminated string buffer which will contain the message date. This parameter cannot be NULL. An alternate version of this method accepts a reference to a **CString** object which will contain the date and time when the method returns.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. This parameter cannot be zero.

bLocalize

An optional boolean flag which specifies if the date and time should be localized for the current timezone. If this parameter is omitted, the date and time will not be localized.

Return Value

If the method succeeds, the return value the length of the date and time string, not including the terminating null character. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDate** method copies the formatted date string into the buffer provided by the caller. If the message contains a valid `Date` or `Sent` header, the function retrieves and, if requested, localizes the date and time string before copying it to the specified buffer. The returned string always conforms to the standard format described in RFC 822. To convert this date string into a numeric value suitable for time calculations, use the **ParseDate** method.

Previous versions of this method returned a pointer to a string managed by the internal message context which could be invalidated or overwritten by subsequent operations. This version ensures the date string remains valid for the lifetime of the caller's buffer, avoiding issues with internal pointer reuse or invalidation that could occur in multithreaded or reentrant scenarios.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatDate, GetHeader, ParseDate, SetDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetExportOptions Method

```
BOOL GetExportOptions(
   LPDWORD lpdwOptions
);
```

The **GetExportOptions** method returns a bitmask that describes the default message export options.

Parameters

lpdwOptions

Pointer to mask of attribute options. The mask is a combination of the following values:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is 0. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Export, Import, SetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetFileType Method

```
UINT GetFileType(
   LPCTSTR lpszFileName,
   LPTSTR lpszContentType,
   INT cchContentType
);

UINT GetFileType(
   LPCTSTR lpszFileName,
   CString& strContentType
);
```

The **GetFileType** method returns the content type for the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file for which content type information is returned.

lpszContentType

Pointer to a string buffer that will contain the MIME type for the specified file. This may be a NULL pointer, in which case this parameter is ignored. If a buffer is provided, it is recommended that it be at least 64 characters in length.

cchContentType

An integer which specifies the maximum number of characters, including the terminating null character, which may be copied into the string buffer.

Return Value

If the method succeeds, the return value is the content type of the specified file. If the method fails, the return value is MIME_CONTENT_UNKNOWN. To get extended error information, call **GetLastError**.

The following values may be returned by this method:

Value	Description
MIME_CONTENT_UNKNOWN	The file content type is unknown. This value may be returned if the message handle is invalid, or if the file extension is unknown and the file could not be opened for read access.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only

	printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

Remarks

The content type for a given file is determined based on the file name extension, or if the extension is not recognized, the actual contents of the file. On 32-bit platforms, the system registry is used to determine the default content type values for a given extension. In all cases, file types that are explicitly set using the **SetFileType** method will override the default system values.

The content type string which is copied to the string buffer is the standard MIME content type description, which specifies a primary type and a subtype, separated by a slash. For example, a plain text file would have a content type of text/plain, while an HTML document would have a content type of text/html. Binary files may be associated with a specific application. For example, the content type for a Microsoft Word document is application/msword. Those binary files which are not associated with a specific application, or have an unrecognized file name extension, have a content type of application/octet-stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachData, AttachFile, SetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetFilteredText Method

```
LONG GetFilteredText(
   LPTSTR lpszFiltered,
   LONG nMaxLength,
   DWORD dwOptions
);
```

The **GetFilteredText** method returns the filtered contents of a HTML or plain text message.

Parameters

lpszFiltered

A null-terminated string buffer which will contain the filtered text when the method returns. This buffer must be large enough to contain all of the filtered content and a terminating null character. If this parameter is NULL, the filtered text will not be copied. This can be used to determine the amount of filtered text that would be returned. This can be used to determine how much memory should be allocated for the filtered text; however, remember to allocate additional space for the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the maximum length is not large enough to store all of the filtered text and a terminating null character, the method will return an error. This value may be zero which indicates you want determine the length of the filtered text without copying any characters.

dwOptions

An unsigned integer which specifies one or more bit flags. which controls what content is filtered from the message. It may be one or more of the following values.

Value	Description
MIME_FILTER_DEFAULT	Perform default filtering which removes HTML markup and decodes HTML entities. No additional filtering of URLs or text is performed.
MIME_FILTER_REMOVE_QUERY	Remove query parameters from hyperlinks such as `?id=123`. Useful for stripping tracking information or session data.
MIME_FILTER_REMOVE_CREDENTIALS	Remove embedded user credentials in URLs such as `user:pass@example.com`, preventing obfuscation or accidental disclosure.
MIME_FILTER_REMOVE_LINKS	Remove all hyperlinks from the message content. This includes both anchor tags in HTML and plain text URLs.
MIME_FILTER_REMOVE_UNICODE	Strip all non-ASCII (Unicode) characters from the message. Use this when only 7-bit clean output is required.
MIME_FILTER_REJECT_LOCAL	Reject any link that refers to internal or local domains, including `.local`, `.lan`, and `localhost`.
MIME_FILTER_REJECT_IP	Reject links that use raw IP addresses instead of

	hostnames, such as `http://192.168.1.1`.
MIME_FILTER_REJECT_PUNYCODE	Reject hyperlinks using Punycode-encoded domain names, such as those with the `xn--` prefix. This helps prevent spoofing using visually similar characters.
MIME_FILTER_REJECT_HTTP	Reject any hyperlinks that use `http://` instead of the secure `https://` protocol.
MIME_FILTER_REJECT_UNSAFE	Reject links using unsafe schemes such as `file:`, `data:`, or `javascript:`. These links are excluded by default unless overridden.
MIME_FILTER_ALLOW_PUNYCODE	Explicitly permit Punycode domain names in URLs even when `MIME_FILTER_REMOVE_UNICODE` is specified. Use when legitimate IDN domains are expected.
MIME_FILTER_ALLOW_UNSAFE	Allow potentially unsafe schemes such as `file:` or `javascript:` in message content. Use with caution and only when the message source is fully trusted.
MIME_FILTER_ALLOW_SYMBOLS	Allow Unicode symbols such as emojis and pictographs in the filtered text.
MIME_FILTER_ALLOW_CONFUSABLES	Allow Unicode characters which can be confused with normal ASCII characters. Use only when the message source is fully trusted.

Return Value

The method will return the number of characters of filtered text copied to the string buffer, not including the terminating null character. If the *lpszFilteredText* parameter is NULL, the method will return the length of the filtered text without copying any characters. A return value of zero indicates the message part does not contain any message text. If the method fails it will return MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **GetFilteredText** method filters MIME-encoded message parts by removing HTML tags and other potentially unsafe or undesired content. It handles both HTML and plain text content types. For HTML, the method converts the markup to plain text and decodes HTML entities. For plain text, it scans for embedded URLs and applies filtering based on the specified options.

When filtering HTML markup from message content, this method:

- Removes script, style, and other non-visible or unsafe elements.
- Converts common tag structures, such as lists and headers, into readable plain text.
- Extracts and optionally filters or removes hyperlinks and image alt text.
- Decodes common HTML entities such as `&trade` and `&copy`, as well as numeric entites like `&#x20.`
- Normalize Unicode characters to their canonical values (NKFD normalization).
- Replace confusable homoglyhs with their ASCII counterparts.

- Remove symbols such as emojis and pictographs which can confuse pattern matching.

The method is designed to produce clean, human-readable output suitable for previewing message contents in text-only environments, logging, or performing further analysis. It ensures that embedded credentials, tracking links, or insecure resources are removed or sanitized based on caller preferences.

By default, the method does not remove Unicode characters or strip links unless explicitly instructed via flags. The `MIME_FILTER_STRICT` constant is a recommended combination for maximum safety and removes potentially malicious or misleading elements from messages. Characters that are visually indistinguishable from standard Latin letters, such as the Cyrillic "a" (U+0430) and Latin "a" (U+0061), will be converted to their ASCII equivalent, preventing homoglyhph attacks and attempts to obsfucate links or defeat pattern matching.

To obtain the original message content, use the **GetText** method.

Example

```
// Search a multipart message for the HTML version of the message
// and write the filtered content to standard output

if (!mailMessage.Open(pszFileName))
{
    _tprintf(_T("Unable to open message %s\n"), pszFileName);
    return;
}

// Get the number of parts in the message
INT nPartCount = mailMessage.GetPartCount();

for (INT nMessagePart = 0; nMessagePart < nPartCount; nMessagePart++)
{
    // Set the current message part
    mailMessage.SetPart(nMessagePart);

    if (mailMessage.IsContentType(_T("text/html")))
    {
        DWORD dwOptions = MIME_FILTER_STRICT;

        // Determine the amount of memory we need to allocate
        LPTSTR pszFiltered = NULL;
        LONG cchFiltered = mailMessage.GetFilteredText(NULL, 0, dwOptions);

        if (cchFiltered < 1)
        {
            // Unable to filter this message part even though its identified
            // as containing HTML or it does not contain any text
            break;
        }
        else
        {
            // Allocate memory for the filtered text; note the extra space is
            // included for the terminating null character
            pszFiltered = (LPTSTR)malloc((cchFiltered + 1) * sizeof(TCHAR));

            if (pszFiltered == NULL)
            {
                // Unable to allocate memory for the filtered text
                break;
            }
```

```
            // Return the filtered message contents in pszFiltered
            cchFiltered = mailMessage.GetFilteredText(
                                pszFiltered,
                                cchFiltered + 1,
                                dwOptions);

            if (cchFiltered > 0)
            {
                // Write the filtered text to standard output on the console
                fwrite(pszFiltered, sizeof(TCHAR), cchFiltered, stdout);
            }

            // Free the memory we allocated for the filtered text
            free(pszFiltered);
            break;
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetContentType, IsContentType, CompareText, GetText, GetTextLength

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetFirstHeader Method

```
BOOL GetFirstHeader(
   LPTSTR lpszHeader,
   INT cchMaxHeader,
   LPTSTR lpszValue,
   INT cchMaxValue
);

BOOL GetFirstHeader(
   CString& strHeader,
   CString& strValue
);
```

The **GetFirstHeader** method returns the header field name and value for the first header in the current message part. This method is typically used in conjunction with **GetNextHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

lpszHeader

A null-terminated string buffer that will contain the name of the first header in the current message part. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A null-terminated string buffer that will contain the value of the first header in the current message part. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is non-zero. If no headers exist for the current message part, or the handle to the message is invalid, the method will return zero. To get extended error information, call **GetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **SetPart** method.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHeaders, GetPart, GetNextHeader, ParseHeader, SetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetHandle Method

```
HMESSAGE GetHandle();
```

The **GetHandle** method returns the message handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message handle associated with the current instance of the class object. If there is no current message, the value INVALID_MESSAGE will be returned.

Remarks

This method is used to obtain the message handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetHeader Method

```
INT GetHeader(
   INT nMessagePart,
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxValue
);

INT GetHeader(
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxValue
);

INT GetHeader(
   INT nMessagePart,
   LPCTSTR lpszHeader,
   CString& strValue
);

INT GetHeader(
   LPCTSTR lpszHeader,
   CString& lpszValue
);
```

The **GetHeader** copies the value of the specified header into a string buffer.

Parameters

nMessagePart

An integer value which specifies which part of the message to return the header value from. A value of zero returns a header value from the main message header, while a value greater than zero returns the header value from that specific part of a multipart message. A value of -1 specifies that the header value should be returned from the current message part. If this argument is omitted, the value will be returned from the current message part.

lpszHeader

A null-terminated string which specifies the message header.

lpszValue

A null-terminated string buffer which will contain the value of the specified header when the method returns. If this parameter is NULL, the method will return the length of the header value without copying the data. This is useful for determining the length of a header value so that a string buffer can be allocated and passed to a subsequent call to the method. In alternate forms of this method, a **CString** object may be specified which will contain the header field value when the method returns.

nMaxValue

An integer value which specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null byte. If the ***lpszValue*** parameter is NULL, the return value is the length

of the header value. If the header field does not exist, a value of zero is returned. If an invalid pointer or message part is specified, a value of MIME_ERROR is returned. To get extended error information, call **GetLastError**.

Example

```
CMailMessage *pMessage = new CMailMessage();
CString strSubject;

// Compose a test message
pMessage->Compose(_T("Bob Jones <bob@example.com>"),
                        _T("Tom Smith <tom@example.com>"),
                        NULL,
                        _T("This is a test message"),
                        _T("This is a test, this is only a test."));

// Change the subject of the message
if (pMessage->GetHeader(0, _T("Subject"), strSubject))
{
    strSubject = _T("Re: ") + strSubject;
    pMessage->SetHeader(0, _T("Subject"), strSubject);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHeaders, GetFirstHeader, GetHeader, GetPart, GetNextHeader, ParseHeader, SetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetLinks Method

```
LONG GetLinks(
   LPTSTR lpszLinkList,
   LONG nMaxLength,
   LONG *lpnLinkCount
   DWORD dwLinkTypes,
);

LONG GetLinks(
   CString &strLinkList,
   LONG *lpnLinkCount
   DWORD dwLinkTypes,
);

INT GetLinks(
   CStringArray &arrLinks,
   DWORD dwLinkTypes
);

INT GetLinks(
   CSimpleArray<CString> &arrLinks,
   DWORD dwLinkTypes
);
```

The **GetLinks** method returns the hyperlinks in the current message part.

Parameters

lpszLinkList

A null-terminated string buffer which will contain all of the matching links found in the current message part. This buffer must be large enough to contain the entire list, including the terminating null character. If you wish to determine the number of characters required to store the list, this parameter can be NULL and the *nMaxLength* parameter should be zero. If you are dynamically allocating memory for the list, make sure you are including additional space for the terminating null. If multiple links are found in the message contents, each link will be separated with a single space. An alternate version of this method accepts a reference to a **CString** object if the project uses MFC or ATL.

arrLinks

An alternate version of this method accepts a reference to a **CStringArray** object in MFC or a **CSimpleArray** object in ATL. In this case, the list of links will be split and returned as a collection of **CString** objects. Note that this version of the method does not return the total length of the link list. Instead, it returns the number of links added to the array.

nMaxLength

The maximum number of characters which can be copied into the string buffer. The buffer must be large enough to store the complete list of links and the terminating null character. If this parameter is zero, the list of links are not copied to the buffer, but the method will return the number of characters required to store the complete list of links, not including the terminating null. If the *lpszLinkList* parameter is NULL, this parameter must be zero.

lpnLinkCount

A pointer to an optional integer which will contain the number of matching links extracted from the current message part. If this information is not required, this parameter can be NULL. This value will

be initialized to zero and updated with the link count if the method is successful.

dwLinkTypes

An unsigned integer value which specifies one or more bitflags which identify what types of links you would like to extract. To extract all supported link types, omit this parameter or use a value of **0**.

Value	Description
MIME_LINK_HTTP	Return standard **http://** links in the current message part.
MIME_LINK_HTTPS	Return secure **https://** links in the current message part.
MIME_LINK_FTP	Return standard **ftp://** links in the current message part.
MIME_LINK_FTPS	Return secure **ftps://** and **ftpes://** links in the current message part.
MIME_LINK_SFTP	Return secure **sftp://** links in the current message part.
MIME_LINK_FILE	Return local **file://** links in the current message part.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer provided. If the *nMaxLength* parameter is zero, the return value specifies the number of characters required to store the list of links, not including the terminating null character. If you use this method to determine the required buffer size, allocate memory for the returned character count plus one additional character for the terminating null. If the method fails, it will return zero. To get extended error information, call the **GetLastError** method. The last error code will be set to NO_ERROR if the parameters are valid but there are no links found in the current message part.

If you use the version of this method which returns the list of links in a **CString** collection, the return value will be the number of links added to the collection. These versions of this method are only available to applications which are built with support for the Microsoft Foundation Classes (MFC) or Active Template Library (ATL).

Remarks

The **GetLinks** method enables you to obtain a list of the hyperlinks which are found in the current message part. The *dwLinkTypes* parameter allows you to specify the types of links you would like to extract. The message contents are decoded and converted to UTF-8 before searching for matching links. If the message content uses HTML markup, it is possible for this method to return links in the HTML header as well as the body. Messages which are composed using Outlook will typically have links to XML namespaces like **http://www.w3.org/TR/REC-html40**. You can omit these links by not including **MIME_LINK_HTTP** in the link type bitmask.

This method can only be used to extract links from message text and cannot be used with file attachments or encoded binary data. For example, if the current message part contains encoded image data, this method will return zero. This method ignores any links that use URL schemes not listed above, such as **mailto:** or **data:**. If you have a need to find a different type of link, you can use the **FindText** method to search the message using a regular expression that matches your specific requirements.

Example

```
if (!mailMessage.Open(TEST_HTMLMSG)
{
     // Unable to open the message file
```

```
     return;
}

// Get the number of parts in a multipart message; if there is only
// one part then this is an simple RFC 822 formatted message
INT nPartCount = mailMessage.GetPartCount();

for (INT nMessagePart = 0; nMessagePart < nPartCount; nMessagePart++)
{
    // Set the current message part
    mailMessage.SetPart(nMessagePart);

    // If this message part is not plain text or HTML, move to the next
    if (mailMessage.GetContentType() != MIME_CONTENT_TEXT)
        continue;

    // Populate an ATL CSimpleArray<CString> object with any links
    CSimpleArray<CString> arrLinks;
    INT nLinkCount = mailMessage.GetLinks(arrLinks, MIME_LINK_HTTPS);

    if (nLinkCount > 0)
    {
        _tprintf(_T("Message part %d has %d secure links\n"), nMessagePart,
nLinkCount);

        // Print each of the links in this message part
        for (INT nIndex = 0; nIndex < nLinkCount; nIndex++)
            _tprintf(_T("\tLink %d: %s\n"), nIndex + 1, (LPCTSTR)arrLinks[nIndex]);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindText, GetContentType, GetFilteredText, GetText, Open, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetNextHeader Method

```
BOOL GetNextHeader(
   LPTSTR lpszHeader,
   INT cchMaxHeader,
   LPTSTR lpszValue,
   INT cchMaxValue
);

BOOL GetNextHeader(
   CString& strHeader,
   CString& strValue
);
```

The **GetNextHeader** method returns the header field name and value for the next header in the current message part. This method is typically used in conjunction with **GetFirstHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

lpszHeader

A null-terminated string buffer that will contain the name of the next header in the current message part. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A null-terminated string buffer that will contain the value of the next header in the current message part. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is non-zero. If no more headers exist for the current message part, or the handle to the message is invalid, the method will return zero. To get extended error information, call **GetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **SetPart** method.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined, with the following exception:

If an imported message has multiple Received headers, then those headers will be returned by **GetNextHeader** in the order in which they appeared in the original message. Note that if **GetHeader** is used to retrieve the Received header, the first Received header in the message will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHeaders, GetFirstHeader, GetPart, ParseHeader, SetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetPart Method

```
INT GetPart();
```

The **GetPart** method returns the current message part index for the specified message.

Parameters

None.

Return Value

If the method succeeds, the return value is the message part index. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CreatePart, DeletePart, GetPartCount, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetPartCount Method

```
INT GetPartCount();
```

The **GetPartCount** method returns the total number of message parts for the specified message. Each message consists of at least one part.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of message parts. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CreatePart, GetPart, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetSender Method

```
INT GetSender(
   LPCTSTR lpszSender,
   INT nMaxLength
);

INT GetSender(
   CString& strSender
);
```

The **GetSender** method returns the email address of the message sender in the specified string buffer.

Parameters

lpszSender

A null-terminated string buffer that will contain the email address of the message sender when the method returns. In alternate forms of this method, a **CString** object may be specified which will contain the email address when the method returns.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. It is recommended that the maximum length be at least 64 characters.

Return Value

If the method succeeds, the return value is the number of bytes copied to the string buffer. If an error occurs, the method will return MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method attempts to determine the email address of the sender that originated the message. It will first check for the presence of a Sender or X-Sender header value. If these headers are not defined, it will use the value of the From header field. It will only return successfully if a valid email address can be found.

If the method succeeds, the string buffer that is provided will only contain an email address. It will not contain the display name of the user associated with the address or any extraneous comments that are included in quotes or parenthesis.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumHeaders, GetFirstHeader, GetHeader, GetPart, GetNextHeader, ParseHeader, SetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetSize Method

```
DWORD GetSize(
   DWORD dwOptions
);
```

The **GetSize** method returns the size of the complete message in bytes.

Parameters

dwOptions

An optional unsigned integer value which specifies how the size of the message should be calculated, based on what header fields should be included. These are the same options used when exporting a message to a file or memory buffer. The following values may be combined using a bitwise Or operator:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.

Return Value

If the method succeeds, the return value is the size of the current message in bytes. If an error occurs, the method will return MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns the size of the complete message, including all headers, the message body and any attachments. It can be used to determine the minimum amount of memory that should be allocated to export the message to a memory buffer using the **Export** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

Export, GetText, GetTextLength, Import, SetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetText Method

```
LONG GetText(
   LPTSTR lpszMessage,
   LONG nMaxLength,
   LONG nOffset
);

LONG GetText(
   CString& strMessage,
   LONG nOffset
);
```

The **GetText** method copies the text of the current message part to the specified buffer.

Parameters

lpszMessage

A null-terminated string buffer that will receive a copy of the message text when the method returns.

nMaxLength

The maximum number of bytes to copy into the buffer. The size of the buffer provided must be larger than the content length for the current message part.

nOffset

An optional parameter which specifies the byte offset from the beginning of the message. A value of zero specifies the first character in the body of the message part.

Return Value

If the method succeeds, the return value is number of bytes copied into the buffer. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

If your project targets a multi-byte character set, this method will always return the message contents as UTF-8 text, regardless of the original character set specified in the message itself. This ensures that characters in the original text are preserved, regardless of the default ANSI code page on the local system. It is recommended you build your project to use Unicode whenever possible. If your application must use ANSI, you can use the **LocalizeText** method to convert the Unicode text to a specific character set.

You should not determine the maximum size of the output buffer using the **GetContentLength** method. That method returns the content size in bytes as it is stored in the message, and does not account for any character encoding or Unicode conversion which may be required. The content length can be used to estimate the amount of text stored in the message part, but you should always allocate a buffer which is larger than the length specified in the message. It is recommended you use the **GetTextLength** method to detemrine how large the string buffer should be.

If the *nMaxLength* parameter does not specify a buffer size large enough to store the contents of the current message part, this method will fail and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. Your application must ensure the buffer is large enough to contain the complete message text and a terminating null character. If you wish to determine the actual length of the decoded message text, you can specify the *lpszMessage* parameter as NULL and *nMaxLength* as zero. This will cause the method to return the number of characters in the current message part,

not including the terminating null. When allocating a buffer on the heap, make sure you include additional space for the terminating null character. See the example below.

If the current message part contains HTML markup and you want to display it as plain text to the user, you can use the **GetFilteredText** method. It can remove markup from the message text, as well as remove potentially dangerous hyperlinks.

Example

```
if (mailMessage.GetContentType(hMessage) == MIME_CONTENT_TEXT)
{
    // Get the size of the decoded/converted message text
    LONG cchMessage = mailMessage.GetTextLength();
    if (cchMessage < 1)
    {
        _tprintf(_T("Message part %d does not contain any text\n"), nMessagePart);
        return;
    }

    // Allocate a string buffer for the message text +1 for the null character
    LPTSTR pszMessage = (LPTSTR)malloc((cchMessage + 1) * sizeof(TCHAR));
    if (pszMessage == NULL) // Virtual memory exhausted
        return;

    // Copy the message contents to the buffer
    cchMessage = mailMessage.GetText(pszMessage, cchMessage + 1);
    if (cchMessage > 0)
    {
        // Write the contents of the message part to the console
        fwrite(pszMessage, sizeof(TCHAR), cchMessage, stdout);
        fflush(stdout);
    }

    // Free the memory allocated for our text buffer
    free(pszMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AppendText, ClearText, CompareText, Export, GetContentLength, GetTextLength, Import, LocalizeText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetTextLength Method

```
LONG GetTextLength();
```

Parameters

None.

Return Value

This method returns the number of text characters in the current message part. If the current message part does not contain any text, the method will return zero. If the method fails it will return MIME_ERROR. To retrieve extended error information, call the **GetLastError** method.

Remarks

The **GetTextLength** method returns the actual length of the message text in the current message part. This value may be different than what is returned by the **GetContentLength** method, which returns the value of the content length in bytes, as specified in the message header. The **GetTextLength** method returns the number of characters (not bytes) in the message text after the contents have been decoded and converted to Unicode.

Example

```
if (mailMessage.GetContentType(hMessage) == MIME_CONTENT_TEXT)
{
    // Get the size of the decoded/converted message text
    LONG cchMessage = mailMessage.GetTextLength();
    if (cchMessage < 1)
    {
        _tprintf(_T("Message part %d does not contain any text\n"), nMessagePart);
        return;
    }

    // Allocate a buffer for the message text +1 for the null character
    LPTSTR pszMessage = (LPTSTR)malloc((cchMessage + 1) * sizeof(TCHAR));
    if (pszMessage == NULL) // Virtual memory exhausted
        return;

    // Copy the message contents to the buffer
    cchMessage = mailMessage.GetText(pszMessage, cchMessage + 1);
    if (cchMessage > 0)
    {
        // Write the contents of the message part to the console
        fwrite(pszMessage, sizeof(TCHAR), cchMessage, stdout);
        fflush(stdout);
    }

    // Free the memory allocated for our text buffer
    free(pszMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

IsContentType, GetFilteredText, GetContentLength, GetHeader, GetText, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::Import Method

```
BOOL Import(
   LPCTSTR lpszFileName
);

BOOL Import(
   LPCTSTR lpszMessage,
   INT nLength
);
```

The **Import** method imports the message from a file or a string buffer in memory, replacing the current message contents.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file that contains the message to be imported.

lpszMessage

A null-terminated string which contains the message to be imported.

nLength

An integer value which specifies the number of bytes to read from the string buffer. If this argument is -1, then the entire contents of the string up to the terminating null character will be imported.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method replaces the current contents of the message with the contents of the specified file or string buffer. The file must be a text file which uses the standard MIME message structure. It may contain multiple parts, including file attachments and alternative representations of the same message. This function cannot be used to open or parse messages stored in a binary format.

The maximum size of an individual message is 2 GiB (approximately 2 billion characters). This limitation is implemented as a safety check to prevent importing very large files or large blocks of memory which are unlikely to contain a valid MIME formatted message. Most services impose much lower limits on the maximum size of an individual mail message.

If another message was previously opened or created using the same class instance, the previous contents will be cleared prior to importing the new message. If you wish to preserve any changes made to the current message, use the **Export** method prior to calling this method.

The version of this method which accepts a file name is functionally similar to the **Open** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Export, Open, ParseText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::IsContentType Method

```
BOOL IsContentType(
   LPCTSTR lpszContentType
);

BOOL IsContentType(
   INT nMessagePart,
   LPCTSTR lpszContentType
);
```

The **MimeIsContentType** method checks if the content type for the message matches the specified content type.

Parameters

hMessage

Handle to the message.

nMessagePart

An optional integer value which specifies which part of the message to compare against the content type. A value of zero returns compares the content type against the main message header, while a value greater than zero compares the content type against the content type for a specific part of a multipart message. If this parameter is omitted or a value of MIME_CURRENT_PART (-1) is specified, the content type will be compared against the current message part.

lpszContentType

Pointer to a string which specifies the content type to compare against. This parameter cannot be NULL and it must specify a valid MIME media type. It is permitted to omit the media subtype, in which case the content type match will only be made against the base type. For example, both `"application/pdf"` and `"application"` are permitted as valid content types. Note that this is a prefix match, not a wildcard pattern match.

Return Value

The method will return a non-zero value if the specified content type matches the content type for the message part. If the content types do not match or one of the parameters is invalid, the method will return zero. To get extended error information, call **GetLastError**. If the parameters are valid but the content types do not match, the last error code will be set to NO_ERROR.

Remarks

The **IsContentType** method is used to check the content type for a specific part of a MIME-encoded message against a known media type. Multipurpose Internet Mail Extensions (MIME) defines a standard way of classifying the content of an email message or its individual parts, such as plain text, HTML, images, or file attachments.

A MIME media type, also referred to as a content type, is typically expressed in the form `type/subtype`. For example, the content type `text/plain` identifies unformatted plain text, while `text/html` identifies HTML formatted text. The content type for each message part is typically specified in its `Content-Type` header field. This method performs a case-insensitive comparison between the message part's declared content type and the type provided by the caller. It does not consider optional parameters, such as character encoding or boundary markers. For example, a content type of `"text/html; charset=UTF-8"` will match `"text/html"`.

The following table lists some of the most common MIME media types used in email messages:

Media Type	Description
`text/plain`	Plain text with no formatting
`text/html`	HTML-formatted text, commonly used for rich email content
`multipart/alternative`	Message that contains multiple representations of the same content, such as both plain text and HTML
`multipart/mixed`	Message that includes a combination of text and attachments
`multipart/related`	Message that includes related resources, such as inline images referenced by HTML content
`image/jpeg`	JPEG image
`image/png`	PNG image
`application/pdf`	PDF document
`application/octet-stream`	Generic binary data, typically used for file attachments

To obtain the full content type string for a message part, including any optional parameters, use the **GetHeader** method with the header name `"Content-Type"`.

Example

```
// Search the message for a PDF attachment by checking the content type
// for each part of a multipart message.
//
// If the content type matches "application/pdf" then that message part
// should contain a PDF attachment. We prompt the user where to store
// the file and then exit the loop.

INT nParts = mailMessage.GetPartCount();

for (INT nMessagePart = 1; nMessagePart < nParts; nMessagePart++)
{
    if (mailMessage.SetPart(nMessagePart) == MIME_ERROR)
        break;

    if (mailMessage.IsContentType(_T("application/pdf")))
    {
        TCHAR szAttachment[MAX_PATH] = _T("");

        // Get the name of the file attached to the message. This can be
        // useful for display purposes, but exercise caution if you want
        // to use that name to store the file locally.
        if (mailMessage.GetAttachedFileName(szAttachment, MAX_PATH))
        {
            _tprintf(_T("Found attachment \"%s\" in part %d\n"), szAttachment,
nMessagePart);

            // The recommended approach is to prompt your user for where
            // they would like you to store store the file attachment;
            // avoid using the remote filename directly without validating
            // or prompting the user.
            TCHAR szLocalFile[MAX_PATH] = _T("");

            OPENFILENAME ofn = {0};
```

```
            ofn.lStructSize = sizeof(OPENFILENAME);
            ofn.hwndOwner = NULL; // No owner window
            ofn.lpstrFilter = "Text Files (*.txt)\0*.txt\0All Files (*.*)\0*.*\0";
            ofn.lpstrFile = szLocalFile;
            ofn.nMaxFile = MAX_PATH;
            ofn.lpstrTitle = "Select File to Save";
            ofn.Flags = OFN_OVERWRITEPROMPT;

            // If the user cancels the dialog, we are finished
            if (!GetSaveFileName(&ofn))
                 break;

            // Extract the attachment to the file name specified by the user
            if (!mailMessage.ExtractFile(szLocalFile))
            {
                DWORD dwError = mailMessage.GetLastError();
                _tprintf(_T("Failed to extract file (Error 0x%08lX)\n"), dwError);
            }
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareText, ExtractFile, FindAttachment, GetAttachedFileName, GetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key the initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CMailMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::LocalizeText Method

```
LONG LocalizeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   LPBYTE lpOutput,
   LONG cbOutput,
   UINT nCharacterSet
);
```

The **LocalizeText** method converts a Unicode string to its ANSI equivalent using the specified character set.

Parameters

lpszInput

A null-terminated string which contains the Unicode text which should be localized. If the ANSI version of this function is called, the input text must be in UTF-8 format or the function will fail. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be localized. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpOutput

A buffer which will contain the localized ANSI version of the input text. This parameter cannot be a NULL pointer.

cbOutput

An integer value which specifies the maximum number of bytes which can be copied into the output buffer. The buffer must be large enough to store all of the localized text. This value must be greater than zero.

nCharacterSet

An optional numeric identifier which specifies the character set to use when localizing the text. If this parameter is omitted, the locale for the current thread will be used when localizing the text.

Return Value

If the input text can be successfully localized, the return value is the number of bytes copied into the output buffer. If the function returns zero, then no text was localized. If the function fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **LocalizeText** method enables the application to localize a Unicode string, returning the ANSI version of that text using the specified character set. Because library handles all text as Unicode internally, the ANSI functions in this library will always return UTF-8 encoded text. This method allows you to easily convert that UTF-8 text to another supported character set.

If the ANSI version of this function is called, the input text must use UTF-8 character encoding or the method will fail with the last error set to ST_ERROR_INVALID_CHARSET.

If the *nCharacterSet* parameter is MIME_CHARSET_DEFAULT or MIME_CHARSET_UNKNOWN the input text will be converted to the default ANSI code page for the current thread locale. If there are characters in the Unicode input text which cannot be converted to an ANSI equivalent using the

specified character set, those characters will be replaced by the default character for your locale, typically a question mark symbol. You cannot specify MIME_CHARSET_UTF16 as the character set.

This method will always attempt to ensure that the output buffer is terminated with an extra null byte so it is easier to work with as a standard C-style null-terminated string. However, if the output buffer is not large enough to accommodate the extra null byte, it will not be copied. It is always recommended that your output buffer be somewhat larger than the length of the original input text to account for multi-byte character sequences. If the output buffer is not large enough to contain the entire localized text, no bytes will be copied to the output buffer and the function will fail with the last error set to ST_ERROR_BUFFER_TOO_SMALL.

This function is only required if your application needs to localize the UTF-8 text returned by another function and convert it to a specific 8-bit ANSI character set. For example, if you have an application which calls the ANSI version of **MimeGetMessageText**, it will return the message contents as UTF-8 text. If you need to display that text as localized ANSI, you can call this function to convert the UTF-8 text to your current locale or a specific character set.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DecodeText, EncodeText, GetText, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::NormalizeText Method

```
LONG NormalizeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG nMaxLength
);

LONG NormalizeText(
   LPCTSTR lpszInput,
   LONG cchInput,
   CString& strOutput
);
```

The **NormalizeText** method normalizes the input text to replace confusable Unicode homoglyphs, remove symbols and trim whitespace.

Parameters

lpszInput

A null-terminated string which contains the Unicode text that will be normalized. The input may only contain text and cannot include non-printable control characters. If this parameter is NULL or if the input contains binary data, this method will fail. If the ANSI version of this method is called, the input string must be contain ASCII or UTF-8 encoded Unicode characters.

cchInput

An integer which specifies the number of characters in the input string. If this parameter is -1, the length of the string will be determined by counting the number of characters up to a terminating null character.

lpszOutput

A null-terminated string buffer which will contain the normalized text. The buffer must be large enough to contain the entire string and a terminating null character or the method will fail. If this parameter is NULL and the ***nMaxLength*** parameter is zero, the method will return the number of characters required to store the normalized text, not including the terminating null character. An alternate version of this method accepts a reference to a **CString** object which will contain the normalized text when the method returns.

nMaxLength

The maximum number of characters which can be copied into the ***lpszOutput*** string buffer. If this value is zero, no characters will be copied to the output buffer and the method will return the length of the filtered output, not including the terminating null character. This can be used to determine how much memory should be allocated to store the filtered text.

Return Value

If the method succeeds, it will return the number of characters copied into the output buffer, not including the terminating null character. If the ***lpszOutput*** parameter is NULL and ***nMaxLength*** is zero, the method will return the number of characters in the filtered text, not including the terminating null. If the method fails, or there is no text in the input buffer, the method will return zero.

Remarks

The **NormalizeText** method performs a multi-stage normalization and cleanup of Unicode text, intended to simplify and standardize input that may contain visually similar (confusable) characters,

diacritical marks, or inconsistent whitespace. It applies NFKD (Compatibility Decomposition) normalization converting characters into their canonical forms and then replaces any confusable Unicode characters with their ASCII equivalents based on an internal mapping table. This method is similar to calling the **NormalizeString** function in the Windows API, however it performs additional checks and is more strict with how the input is validated.

If your project is configured to use a multibyte character set (ANSI), the input string is checked to ensure that it only contains valid UTF-8 encoding and will reject strings which contain invalid sequences. For example, it will reject multibyte UTF-8 sequences which are over-long or truncated, invalid continuation sequences or characters outside of valid Unicode code points. You should never provide a localized ANSI string composed using the current code page. If you wish to normalize a localized string, you should convert it to Unicode using the **MultiByteToWideChar** function and then call the Unicode version of this function.

If your project is configured to use Unicode, the input string is validated to ensure that it only contains valid UTF-16 encoding and will reject strings which contain invalid sequences. This includes checking to ensure there are no unpaired surrogates, invalid code points, unprintable control characters or private-use characters in the string. If the input string does not contain valid Unicode character sequences, the method will return zero and the last error code will be set to ST_ERROR_INVALID_UNICODE.

This method helps mitigate issues with characters that are visually indistinguishable from standard Latin letters, such as using the Cyrillic "a" (U+0430) instead of the Latin "a" (U+0061). It also removes symbols (such as emojis, flags and pictographs) which can confuse pattern matching and removes combining marks that can be used to visually modify characters or obscure text. Unicode whitespace is trimmed from both ends of the string, including zero-width and non-breaking spaces.

This method is particularly useful in security-sensitive applications where input normalization is critical, such as sanitizing user-provided domain names, filenames, or login credentials to prevent spoofing, phishing, or bypassing filters through the use of homoglyphs or invisible characters.

If you want a normalized version of the text in a message, you can use the **GetFilteredText** method. It performs many of the same normalizations that this method does, but allows for more flexibility through options which allow you to control how the message text is processed and filtered. It is particularly useful for filtering extraneous HTML from the message, enabling you to display a version of the message that is safer for the user (such as removing unsafe links from the message contents).

This method is deliberately biased toward security rather than readability and can affect the output when applied to text containing legitimate non-Latin characters. It is intended for use on text that is expected to contain only printable ASCII characters. For example, it will filter and normalize text containing certain Greek or Cyrillic characters that could be used to spoof ASCII letters or URL delimiters. While this helps mitigate obfuscated or malicious content, it can also make legitimate text in other languages unreadable.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareText, FindText, GetContentType, GetFilteredText, GetText, IsContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::Open Method

```
BOOL Open(
   LPCTSTR lpszFileName,
   DWORD dwOptions
);
```

The **Open** method opens and reads the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the text file to open. This parameter cannot be NULL.

dwOptions

An integer which specifies options for opening the message file. This parameter is reserved for future use and can be omitted. If specified, it should always be zero in the current implementation.

Value	Description
MIME_OPEN_DEFAULT	The default mode for opening the message file. Currently, this is the only option which should be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails it returns zero. To get extended error information call the **GetLastError** method.

Remarks

This method opens the specified file, reads the contents into memory and then parses the message. The file must be a text file which uses the standard MIME message structure. It may contain multiple parts, including file attachments and alternative representations of the same message. This function cannot be used to open messages stored in a binary format.

If the file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

The maximum size of an individual message is 2 GiB (approximately 2 billion characters). This limitation is implemented as a safety check to prevent opening very large files which are unlikely to contain a valid MIME formatted message. Most services impose much lower limits on the maximum size of an individual mail message.

Note: This method can only be used to open and parse messages structured according to the MIME (Multipurpose Internet Mail Extensions) standard, such as plain text `.eml` or `.txt` files. It cannot open messages stored in proprietary or binary formats used by certain email clients. For example, it does not support Microsoft Outlook message files such as `.msg` or mailbox files like `.pst` and `.ost`, nor does it support formats used by other clients such as Thunderbird's `.msf` (mail summary files) or the mbox family (`.mbx`, `.mbox`) when stored as binary containers. Attempting to open these file types will result in an error. If you need to work with messages exported from Outlook or other proprietary clients, first save or export them in the standard text format before using this method.

If another message was previously opened or created using the same class instance, the internal handle which references the previous message will be closed. If you wish to preserve any changes

made to the current message, use the **Export** method prior to calling this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachFile, Close, Export, ExtractFile, Import

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ParseAddress Method

```
INT ParseAddress(
   LPCTSTR lpszString,
   LPCTSTR lpszDomain,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT ParseAddress(
   LPCTSTR lpszString,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT ParseAddress(
   LPCTSTR lpszString,
   CString& strAddress
);
```

The **ParseAddress** method parses a string for an email address, copying the address to the specified buffer.

Parameters

lpszString

A null-terminated string which contains the email address to parse.

lpszDomain

A null-terminated string which specifies a default domain for the address. This parameter may be NULL or point to an empty string if no default domain is required.

lpszAddress

A null-terminated string buffer which will contain the parsed email address when the method returns. It is recommended that this buffer be at least 128 characters in length.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the address. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ParseAddress** method is useful for parsing the email addresses that may be specified in various header fields in the message. In many cases, the addresses have additional comment characters which are not part of the address itself. For example, one common format used is as follows:

"User Name" <user@domain.com>

In this case, the email address is enclosed in angle brackets and the name outside of the brackets is considered to be a comment which is not part of the address itself. Another common format is:

user@domain.com (User Name)

In this case, there is the address followed by a comment which is enclosed in parenthesis. The

ParseAddress method recognizes both formats, and when passed either string, would return the following address:

user@domain.com

If there was no domain specified in the address, that is just a user name was specified, then the value the ***lpszDomain*** parameter is added to the address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Export, ExtractFile, Import

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ParseDate Method

```
BOOL ParseDate(
   LPCTSTR lpszDate,
   LONGLONG *lpnSeconds,
   LONG *lpnTimezone,
   BOOL bLocalize
);

BOOL ParseDate(
   LPCTSTR lpszDate,
   LONG *lpnSeconds,
   LONG *lpnTimezone,
   BOOL bLocalize
);
```

The **ParseDate** method parses a date string, returning the number of seconds since 1 January 1970 and the difference in seconds between the specified timezone and coordinated universal time. If the string does not specify a timezone, the local timezone is used.

Parameters

lpszDate

A null-terminated string which specifies the date to be parsed. The date string must be in the standard format defined by RFC 822.

lpnSeconds

A pointer to a 64-bit or 32-bit integer which is set to the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C/C++ time methods. This pointer may be NULL, in which case the parameter is ignored. It is recommended you always use the version which accepts a pointer to a 64-bit integer.

lpnTimezone

A pointer to a 32-bit integer which is set to the difference in seconds between the specified date's timezone and coordinated universal time (also known as Greenwich Mean Time). This pointer may be NULL, in which case the parameter is ignored.

bLocalize

A boolean flag which determines if the date should be localized to the current timezone, regardless of the timezone specified in the date string. A non-zero value specifies the timezone for the local system will be used, adjusted for daylight savings time if applicable.

Return Value

If the date could be successfully parsed, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is not a general purpose date parsing function, and may not be capable of parsing dates for a specific locale. The date and time should be in a standard format as outlined in RFC 822, which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **FormatDate** method.

If the date and time does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you use this function to parse input from a

user, because in most cases they will not provide a timezone and will assume the date and time they enter is for their current timezone.

The value of the ***bLocalize*** parameter will only change the number of seconds offset by the current timezone and does not affect the value returned in the ***lpnSeconds*** parameter. If the date is localized, the timezone offset will be adjusted for daylight savings if it was in effect at the time.

Developers are strongly encouraged to use the version of this function which expects a 64-bit integer to represent the number of seconds since the epoch. The version which uses a 32-bit integer limits the representable range to dates before 19 January 2038 (the so-called "Year 2038 problem"). After this date, 32-bit time values will overflow, resulting in incorrect date calculations and potential failures in software that processes email message dates. By contrast, the 64-bit integer value matches the standard `time_t` type used by modern C/C++ libraries. This not only eliminates the Year 2038 limitation but also ensures compatibility with current and future systems, allowing correct handling of dates both far in the past and well into the future. Whenever possible, applications should transition to the 64-bit version of this method to ensure long-term reliability and portability.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatDate, GetDate, SetDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ParseHeader Method

```
BOOL ParseHeader(
   LPCTSTR lpszHeader
);
```

The **ParseHeader** method parses a line of text and adds the header and value to the current message.

Parameters

lpszHeader

Pointer to a string which contains the header and value to be added to the message.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to parse a line of text that is part of a message header. The string must consist of a header name, followed by a colon, followed by the header value. The header name may only consist of printable characters, and may not contain whitespace (space, tab, carriage return or linefeed characters).

In most cases, it is recommended you use the **SetHeader** method to set the value of a header.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, Import, ParseText, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ParseText Method

```
BOOL ParseText(
   LPCTSTR lpszText,
   INT cchText
);
```

The **Parse** method parses the contents of the specified buffer and adds the contents to the message.

Parameters

lpszText

Pointer to a string buffer which contains the text to be added to the message contents.

cchText

The length of the specified buffer. If this value is -1 all of the characters in the string will be parsed up to the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is useful when the application needs to parse an arbitrary block of text and add it to the specified message. If the string contains header fields, the values will be added to the message header. Once the end of the header block is detected, all subsequent text is added to the body of the message.

To open and parse a text file which contains a MIME structured email message, use the **Open** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Clear, Close, Import, Open, ParseHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetDate Method

```
BOOL SetDate(
   LPCTSTR lpszDate,
   BOOL bLocalize
);
```

The **SetDate** method sets the date and time in the header for the specified message.

Parameters

lpszDate

Pointer to a string which specifies the date and time. If this parameter specifies a zero-length string or a NULL pointer, the current date and time will be used.

bLocalize

Boolean flag that specifies the date and time should be localized for the current timezone.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The date string should be in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **FormatDate** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatDate, GetDate, GetHeader, ParseDate, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetExportOptions Method

```
BOOL SetExportOptions(
   DWORD dwOptions
);
```

The **SetExportOptions** method specifies a bitmask that describes current message export options.

Parameters

dwOptions

Mask of attribute options. The mask is a combination of the following values:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

By default, the Received and Return-Path headers are not exported. In addition, the order of the headers in an exported message is undefined. This is reasonable behavior for most mail clients, but may not be appropriate for applications which need access to all of the header fields.

Example

```
pMessage->SetExportOptions(MIME_EXPORT_OPTIONS_ALL |
                           MIME_EXPORT_OPTIONS_KEEP_ORDER);
```

```
if (pMessage->Import(lpszFileName))
{
    // Process the message and make any modifications
    // then write the message back out, preserving all
    // of the headers in their original order
    bResult = pMessage->Export(lpszFileName);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Export, Import, GetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetFileType Method

```
BOOL SetFileType(
   UINT nContentType,
   LPCTSTR lpszSubtype,
   LPCTSTR lpszExtension
);
```

The **SetFileType** method associates a per-message content type with a given file name extension. This association is specific to the message, and is not shared by an other messages that may be opened by the process.

Parameters

nContentType

An identifier which specifies the primary content type. It may be one of the following values:

Value	Description
MIME_CONTENT_UNKNOWN	The default content type for the specified extension should be used. This value should only be used to delete a previously defined content type.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

lpszSubtype

A null-terminated string which specifies the MIME subtype. This parameter may be NULL if the content type association is being deleted.

lpszExtension

A null-terminated string which specifies the file name extension that will be associated with the MIME content type.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call **GetLastError**.

Remarks

The **SetFileType** method allows an application to specify a content type for a given file extension, and is typically used to define custom content types for file attachments. The content type will override any default content types associated with the extension, as well as allow new content types to be defined for application-specific files.

Example

In the following example, the file extension ".dat" is associated with a custom content type, a file is attached to the message and then the custom content type is deleted. Note that because the primary content type designates the file as an application specific (non-text) file, it will be automatically encoded when attached to a message:

```
bResult = pMessage->SetFileType(
    MIME_CONTENT_APPLICATION,
    _T("octet-stream"), _T("dat"));

if (bResult)
{
    bResult = pMessage->AttachFile(lpszFileName, MIME_ATTACH_DEFAULT);
    pMessage->SetFileType(MIME_CONTENT_UNKNOWN, NULL, _T("dat"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AttachData, AttachFile, GetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetHeader Method

```
BOOL SetHeader(
   INT nMessagePart,
   LPCTSTR lpszHeader,
   LPCTSTR lpszValue
);

BOOL SetHeader(
   LPCTSTR lpszHeader,
   LPCTSTR lpszValue
);
```

The **SetHeader** method adds or updates a header field in the specified message.

Parameters

nMessagePart

An integer value which specifies which part of the message the header should be set or modified in. A value of zero sets a header value in the main message header block, while a value greater than zero sets the header value in that specific part of a multipart message. If this argument is omitted or a value of -1 is specified, the header value will be set in the current message part.

lpszHeader

Pointer to a string which specifies the header field that will be added or updated.

lpszValue

Pointer to a string which specifies the value for the header field. This pointer may be NULL, which causes the header field to be removed from the message.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddHeaders, EnumHeaders, GetFirstHeader, GetHeader, GetPart, GetNextHeader, SetPart, SetText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetPart Method

```
INT SetPart(
   INT nNewPart
);
```

The **SetPart** method sets the current message part index for the specified message.

Parameters

nNewPart

The new message part index. A value of zero specifies the main message part.

Return Value

If the method succeeds, the return value is the previous message part index. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

Multipart email messages are structured using the MIME (Multipurpose Internet Mail Extensions) standard, which allows a single email to contain multiple content types, such as plain text, HTML, and file attachments, within a single message body. These components are organized as discrete parts, each with its own headers and body, separated by unique MIME boundaries. In a typical `multipart/alternative` message, part 0 often contains the message preamble (e.g.: "This is a multipart message in MIME format"), part 1 may contain the plain text version of the message, and part 2 may contain the HTML version. However, multipart messages are not required to include both plain text and HTML content. A message may only include an HTML part and one or more attachments, or any other combination of content types. The structure and ordering of parts depends entirely on how the original sender or mail client composed the message.

To simplify navigating these multipart structures, the **SetPart** and **GetPart** methods enable you to iterate through the individual parts of a message. The message parts are zero-based, with index 0 always referring to the top-level message headers and preamble. By calling **SetPart**, you can change the currently selected message part, enabling functions like **GetHeader** or **GetText** to operate on that specific segment. Calling **GetPart** returns the current part index, which can be useful for saving and restoring state while iterating over a message.

The internal MIME parser flattens multipart messages, meaning that if a message contains nested multipart sections (such as a multipart/mixed part containing a multipart/alternative part), all leaf parts are presented in a linear sequence. This simplifies access and avoids the complexity of recursive traversal. You can determine the total number of parts using **GetPartCount**, and then iterate from part 0 to part N-1 to examine or extract each segment in order. This approach provides both fine-grained access to MIME message parts and an intuitive model for parsing even complex message structures.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetPart, GetPartCount, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::SetText Method

```
LONG SetText(
   LONG nOffset,
   LPCTSTR lpszText
);

LONG SetText(
   LPCTSTR lpszText
);
```

The **SetText** method copies the specified text into the body of the current message part.

Parameters

nOffset

The offset into the body of the message part. A value of -1 specifies that the text will be appended to the message body. If this argument is omitted, the current message text is replaced with the contents of the *lpszText* argument.

lpszText

A null-terminated string which specifies the text to be copied to the current message part at the given offset.

Return Value

If the method succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be copied into the current message part. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AppendText, Clear, ClearText, CompareText, GetText, SetHeader, SetPart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMailMessage Character Sets

Value	Name	Code Page	Description
MIME_CHARSET_USASCII	us-ascii	20127	A character set which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered.
MIME_CHARSET_ISO8859_1	iso-8859-1	28591	A character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin-1. This character set is similar to Windows code page 1252 (Windows-1252), however there are differences such as the Euro symbol.
MIME_CHARSET_ISO8859_2	iso-8859-2	28592	A character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin-2. This character set is similar to Windows code page 1250, however the characters are arranged differently.
MIME_CHARSET_ISO8859_3	iso-8859-3	28593	A character set for southern European languages such as Maltese and Esperanto. This character set was also used with the Turkish language, but it was superseded by ISO 8859-9 which is the preferred character set for Turkish. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_4	iso-8859-4	28594	A character set for northern European languages such as Latvian, Lithuanian and Greenlandic. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_5	iso-8859-5	28595	A character set for Cyrillic languages such as Russian, Bulgarian and Serbian. This character set was never widely adopted and most mail messages use

			either KOI8 or UTF-8 encoding.
MIME_CHARSET_ISO8859_6	iso-8859-6	28596	A character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	iso-8859-7	28597	A character set for the Greek language. This character set is also commonly referred to as Latin/Greek. This character set is no longer widely used and has largely been replaced with UTF-8 which provides more complete coverage of the Greek alphabet.
MIME_CHARSET_ISO8859_8	iso-8859-8	28598	A character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. In most cases it is recommended that you use UTF-8 instead of this character set.
MIME_CHARSET_ISO8859_9	iso-8859-9	28599	A character set for the Turkish language. This character set is also commonly referred to as Latin-5. This character set is nearly identical to ISO 8859-1, except that it replaces certain Icelandic characters with Turkish characters.
MIME_CHARSET_ISO8859_10	iso-8859-10	28600	A character set for the Danish, Icelandic, Norwegian and Swedish languages. This character set is also commonly referred to as Latin-6 and is similar to ISO 8859-4.
MIME_CHARSET_ISO8859_13	iso-8859-13	28603	A character set for Baltic languages. This character set is also commonly referred to as Latin-7. This character set is similar to ISO 8859-4, except it adds certain Polish characters and does not support Nordic languages.
MIME_CHARSET_ISO8859_14	iso-8859-14	28604	A character set for Gaelic languages such as Irish, Manx and Scottish Gaelic. This character set is also commonly

			referred to as Latin-8. This character set replaced ISO 8859-12 which was never fully implemented.
MIME_CHARSET_ISO8859_15	iso-8859-15	28605	A character set for western European languages. This character set is also commonly referred to as Latin-9 and is nearly identical to ISO8859-1 except that it replaces lesser-used symbols with the Euro sign and some letters.
MIME_CHARSET_ISO2022_JP	iso-2022-jp	50220	A multi-byte character encoding for Japanese that is widely used with mail messages. This is a 7-bit encoding where all characters start with ASCII and uses escape sequences to switch to the double-byte character sets.
MIME_CHARSET_ISO2022_KR	iso-2022-kr	50225	A multi-byte character encoding for Korean which encodes both ASCII and Korean double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_ISO2022_CN	x-cp50227	50227	A multi-byte character encoding for Simplified Chinese which encodes both ASCII and Chinese double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_KOI8R	koi8-r	20866	A character set for Russian using the Cyrillic alphabet. This character set also covers the Bulgarian language. Most mail messages in the Russian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_KOI8U	koi8-u	21866	A character set for Ukrainian using the Cyrillic alphabet. This character set is similar to the KOI8-R character set, but replaces certain symbols with Ukrainian letters. Most mail messages in the Ukrainian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_GB2312	x-cp20936	20936	A multi-byte character encoding which can represent ASCII and simplified Chinese characters. It has been superseded by GB18030, however it

			remains widely used in China.
MIME_CHARSET_GB18030	gb18030	54936	A Unicode transformation format which can represent all Unicode code points and supports both simplified and traditional Chinese characters. It is backwards compatible with GB2312 and supersedes that character set.
MIME_CHARSET_BIG5	big5	950	A multi-byte character set that supports both ASCII characters and traditional Chinese characters. It is widely used in Taiwan, Hong Kong and Macau. It is no longer commonly used in China, which has developed GB18030 as a standard encoding. Microsoft's implementation of Big5 on Windows does not support all of the extensions and is missing certain code points.
MIME_CHARSET_UTF7	utf-7	65000	A Unicode transformation format that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 when quoted-printable or base64 encoding is not desired.
MIME_CHARSET_UTF8	utf-8	65001	A Unicode transformation format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it is recommended that you use either quoted-printable or base64 encoding to ensure compatibility with mail servers that do not support 8-bit characters.
MIME_CHARSET_UTF16	utf-16le	N/A	A 16-bit Unicode format that represents each character as a 16-bit value in little endian byte order. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead. UTF-16 characters in big endian byte order are not supported.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

CMailMessage, Compose, CreatePart, DecodeText, EncodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Message Store Class Library

Store a collection of email messages on the local system.

Reference

- Class Methods
- Error Codes

Library Information

Class Name	CMessageStore
File Name	CSMSGV12.DLL
Version	12.0.3024.1286
LibID	123A728A-D6CD-435F-A76D-0BA3C07DDFC1
Import Library	CSMSGV12.LIB
Dependencies	None

Overview

The Message Store class library provides an interface for managing a local message storage file that can be used to store and retrieve multiple messages. Functions are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values.

SocketTools also includes a class library for managing individual messages. For more information, refer to the CMailMessage class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Message Store Class Methods

Class	Description
CMessageStore	Constructor which initializes the current instance of the class
~CMessageStore	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified message handle to this instance of the class
CloseFile	Close the current message storage file
CopyFile	Duplicate the contents of the current message store in a new file
DeleteMessage	Remove the specified message from the current message store
DetachHandle	Detach the handle for the current instance of this class
FindMessage	Search for a message in the current message store
GetHandle	Return the message handle used by this instance of the class
GetErrorString	Return a description for the specified error code
GetLastError	Return the last error code
GetMessage	Retrieve a message from the current message store
GetMessageCount	Return the number of messages in the current message store
IsInitialized	Determine if the class has been successfully initialized
OpenFile	Open the specified message storage file
PurgeFile	Purge all deleted messages from the current message store
ReplaceMessage	Replace the specified message in the current message store
SetLastError	Set the last error code
ShowError	Display a message box with a description of the specified error
StoreMessage	Store the specified message in the current message store

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::CMessageStore Method

```
CMessageStore();
```

The **CMessageStore** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

~CMessageStore, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::~CMessageStore

```
~CMessageStore();
```

The **CMessageStore** destructor releases resources allocated by the current instance of the **CMessageStore** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CMessageStore** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. All handles that were client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CMessageStore

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::AttachHandle Method

```
VOID AttachHandle(
   HMESSAGESTORE hStorage
);
```

The **AttachHandle** method attaches the specified message store to the current instance of the class.

Parameters

hStorage

The handle to the message store that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a message store handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that message store.

If a message store handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous message store from being closed, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hStorage*** parameter is presumed to be a valid message store handle and no checks are performed to ensure that the handle is valid. Specifying an invalid message store handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::CloseFile Method

```
BOOL CloseFile(
   BOOL bPurgeMessages
);
```

The **CloseFile** method closes the current message store.

Parameters

bPurgeMessages

An integer value which specifies if deleted messages are purged from the message store. A non-zero value specifies that all messages marked for deletion will be removed from the message store. A value of zero specifies that deleted messages will not be removed from the store.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CloseFile** method closes the storage file that was previously opened, releasing all of the memory allocated for the message store and optionally purging all deleted messages. Note that the current message store will be closed by the class destructor when the class object is deleted or goes out of scope.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CopyFile, OpenFile, PurgeFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::CopyFile Method

```
BOOL CopyFile(
   LPCTSTR lpszFileName
);
```

The **CopyFile** method duplicates the contents of the current message store in a new file.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file that the messages will be copied to. This parameter cannot be NULL and must specify a valid file path and name.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **CopyFile** method is used to create a copy of the current message store in a new file. If the file does not exist, it will be created. If the file already exists, then the contents will be overwritten with the contents of the message store.

Messages that have been marked for deletion are not copied to the new message store file.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFile, OpenFile, PurgeFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::DeleteMessage Method

```
BOOL DeleteMessage(
   LONG nMessageId
);
```

The **DeleteMessage** method removes the specified message from the current message store.

Parameters

nMessageId

An integer value which identifies the message that is to be removed from the message store. Message numbers begin at one and increment for each message in the store.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DeleteMessage** method marks the specified message for deletion from the storage file. When the message store is closed or purged, the message is removed from the file. Once a message has been marked for deletion, it may no longer be referenced by the application. For example, you cannot access the contents of a message that has been deleted.

The message store must be opened with write access. This method will fail if you attempt to delete a message from a storage file that has been opened for read-only access. If the application needs to delete messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CloseFile, FindMessage, GetMessage, GetMessageCount, PurgeFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::DetachHandle Method

```
HMESSAGESTORE DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message store handle associated with the current instance of the class object. If there is no active client session, the value INVALID_MESSAGESTORE will be returned.

Remarks

This method is used to detach a message handle created by the class for use with the SocketTools API. Once the message handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::FindMessage Method

```
LONG FindMessage(
   LONG nMessageId,
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue,
   DWORD dwOptions
);
```

The **FindMessage** method searches for a message in the current message store.

Parameters

nMessageId

An integer value which specifies the message number that should be used when starting the search. The first message in the message store has a value of one.

lpszHeaderName

The string which specifies the name of the header field that should be searched. The header field name is not case sensitive. This parameter cannot be NULL.

lpszHeaderValue

The string which specifies the header value that should be searched for. The search options can be used to specify if the search is case-sensitive, and whether the search should return partial matches to the string. This parameter cannot be NULL.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_SEARCH_DEFAULT	Perform a complete match against the specified header value. The comparison is not case-sensitive.
MIME_SEARCH_CASE_SENSITIVE	The header field value comparison will be case-sensitive. Note that this does not affect header field names. Matches for header names are always case-insensitive.
MIME_SEARCH_PARTIAL_MATCH	Perform a partial match against the specified header value. It recommended that this option be used when searching for matches to email addresses.
MIME_SEARCH_DECODE_HEADERS	Decode any encoded message headers before comparing them to the specified value. This option can increase the amount of time required to search the message store and should only be used when necessary.

Return Value

If the method succeeds, the return value is the number for the message which matches the search criteria. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **FindMessage** method is used to search the message store for a message which matches a specific header field value. For example, it can be used to find every message which is addressed to a specific recipient or has a subject which matches a particular string value.

Example

```
CMessageStore mailStorage;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
LONG nMessageId = 1;

if (! mailStorage.OpenFile(lpszFileName))
{
    // Unable to open the storage file
    return;
}

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = mailStorage.FindMessage(nMessageId,
                                         lpszHeader,
                                         lpszAddress,
                                         MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        CMailMessage mailMessage;

        // Get a copy of the message that was found
        if (mailStorage.GetMessage(nMessageId, mailMessage))
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = mailMessage.Export(szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}

mailStorage.CloseFile();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteMessage, GetMessage, GetMessageCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::GetHandle Method

```
HMESSAGESTORE GetHandle();
```

The **GetHandle** method returns the message store handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message store handle associated with the current instance of the class object. If there is no current message, the value INVALID_MESSAGESTORE will be returned.

Remarks

This method is used to obtain the message store handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGESTORE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::GetMessage Method

```
BOOL GetMessage(
   LONG nMessageId,
   CMailMessage& mailMessage
);

HMESSAGE GetMessage(
   LONG nMessageId,
   DWORD dwOptions
);
```

The **GetMessage** method retrieves a message from the current message store.

Parameters

nMessageId

An integer value which specifies the message number that should be retrieved. The first message in the message store has a value of one.

mailMessage

A **CMailMessage** object which will reference the message that is retrieved from the message store.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
A shared message handle returned by the function. The contents of this message will be overwritten each time this function is called.	
MIME_COPY_STORED_MESSAGE	A message handle is allocated for a copy of the message that is retrieved from the message store.

Return Value

If the first form of the method succeeds the return value is non-zero, otherwise the return value is zero. If the second form of the message succeeds it returns a message handle, otherwise it returns INVALID_MESSAGE. To get extended error information, call **GetLastError**.

Remarks

The second form of the **GetMessage** method returns a message handle for the specified message in the message store. If no options are specified, a temporary message handle is returned that is only valid until the next message is retrieved. If a multithreaded application changes the contents of the temporary message, it will change for all other threads that have obtained a message handle using this function.

If the application must have a unique copy of the message, the MIME_COPY_STORED_MESSAGE option should be specified. Instead of returning a handle to a shared message, the message is duplicated and a handle to that copy of the message is returned. If a reference to a **CMailMessage** object is passed to the first form of this method, this option is used to create a copy of the message.

Example

```
CMessageStore mailStorage;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
LONG nMessageId = 1;

if (! mailStorage.OpenFile(lpszFileName))
{
    // Unable to open the storage file
    return;
}

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = mailStorage.FindMessage(nMessageId,
                                         lpszHeader,
                                         lpszAddress,
                                         MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        CMailMessage mailMessage;

        // Get a copy of the message that was found
        if (mailStorage.GetMessage(nMessageId, mailMessage))
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = mailMessage.Export(szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}

mailStorage.CloseFile();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CMailMessage, DeleteMessage, FindMessage, GetMessageCount, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::GetMessageCount Method

```
BOOL GetMessageCount(
   LPLONG lpnLastMessage
);
```

The **GetMessageCount** method returns the number of messages in the current message store.

Parameters

lpnLastMessage

A pointer to an integer which will contain the message number for the last message in the storage file. If this information is not required, a NULL pointer may be specified.

Return Value

If the method succeeds, the return value is the number of messages in the message store. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageCount** method returns the number of messages in the message store. It is important to note that does not count those messages which have been marked for deletion. This means that the value returned by this function will decrease as messages are deleted.

The message number returned in the ***lpnLastMessage*** parameter will specify the total number of messages in the message store, including deleted messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

DeleteMessage, FindMessage, GetMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key the initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CMessageStore

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::OpenFile Method

```
BOOL OpenFile(
   LPCTSTR lpszFileName,
   DWORD dwOpenMode
);
```

The **OpenFile** method opens the specified message storage file.

Parameters

lpszFileName

A null-terminated string which specifies the name of the storage file.

dwOpenMode

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_STORAGE_READ	The message store will be opened for read access. The contents of the message store can be accessed, but cannot be modified by the process unless it has also been opened for writing.
MIME_STORAGE_WRITE	The message store will be opened for writing. This mode also implies read access and must be specified if the application needs to modify the contents of the message store.
MIME_STORAGE_CREATE	The message store will be created if the storage file does not exist. If the file exists, it will be truncated. This mode implies read and write access.
MIME_STORAGE_LOCK	The message store will be opened so that it may only be accessed and modified by the current process.
MIME_STORAGE_COMPRESS	The contents of the message store are compressed. This option is automatically enabled if a compressed message store is opened for reading or writing.
MIME_STORAGE_MAILBOX	The message store should use the UNIX mbox format when reading and storing messages. This option is provided for backwards compatibility and is not recommended for general use.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **OpenFile** method opens a message storage file which contains one or more messages. If the storage file is opened for read access, the application can search the message store and extract messages but it cannot add or delete messages. To add new messages or delete existing messages from the store, it must be opened with write access.

The message store is designed to be a simple, effective way to store multiple messages together in a single file. When the message store is opened, the contents are indexed in memory. Although there is no specific limit to the number of messages that can be stored, there must be sufficient memory available to build an index of each message and its headers. If the application must store and manage a very large number of messages, it is recommended that you use a database rather than a flat-file message store.

Message Store Format

Each message is prefixed by a control sequence of five ASCII 01 characters followed by an ASCII 10 and ASCII 13 character. The messages themselves are stored unmodified in their original text format. The length of each message is calculated based on the location of the control sequence that delimits each message, and explicit message lengths are not stored in the file. This means that it is safe to manually change the message contents, as long as the message delimiters are preserved.

If the message store is compressed, the contents of the storage file are expanded when the file is opened and then re-compressed when the storage file is closed. Using the MIME_STORAGE_COMPRESS option reduces the size of the storage file and prevents the contents of the message store from being read using a text file editor. However, enabling compression will increase the amount of memory allocated by the library and can increase the amount of time that it takes to open and close the storage file.

The class also has a backwards compatibility mode where it will recognize storage files that use the UNIX mbox format. While this format is supported for accessing existing files, it is not recommended that you use it when creating new message stores or adding messages to an existing store. There are a number of different variants on the mbox format that have been used by different Mail Transfer Agents (MTAs) on the UNIX platform. For example, the mboxrd variant looks identical to the mboxcl2 variant, and they are programmatically indistinguishable from one another, but they are not compatible. For this reason, the use of the mbox format is strongly discouraged.

Example

```
CMailMessage mailMessage;

// Compose a new message
mailMessage.Compose(lpszSender,
                            lpszRecipient,
                            NULL,
                            lpszSubject,
                            lpszMessage,
                            NULL,
                            MIME_CHARSET_DEFAULT,
                            MIME_ENCODING_DEFAULT);

CMessageStore mailStorage;

// Open the message storage file
if (mailStorage.OpenFile(lpszFileName, MIME_STORAGE_WRITE))
{
    // Store a copy of the message in the message store
    nMessageId = mailStorage.StoreMessage(mailMessage);

    if (nMessageId == MIME_ERROR)
    {
        // We were unable to store the message
    }
```

```
    // Close the message store
    mailStorage.CloseFile();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFile, FindMessage, GetMessage, GetMessageCount, PurgeFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::PurgeFile Method

```
BOOL PurgeFile();
```

The **PurgeFile** method purges all deleted messages from the specified message store.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **PurgeFile** method purges all deleted messages from the message store. If the storage file has been opened in read-only mode or there are no messages marked for deletion, this method will take no action.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

CopyFile, DeleteMessage, FindMessage, GetMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::ReplaceMessage Method

```
BOOL ReplaceMessage(
   LONG nMessageId,
   CMailMessage& mailMessage
);

BOOL ReplaceMessage(
   LONG nMessageId,
   HMESSAGE hMessage
);
```

The **MimeReplaceStoredMessage** function replaces the contents of the specified message in a message store.

Parameters

nMessageId

An integer value which specifies the message number that should be replaced.

mailMessage

A **CMailMessage** object which references the message that will be stored.

hMessage

Handle to the message that will be stored.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **ReplaceMessage** method replaces the specified message with a new message. The message number may be a message that has been previously marked for deletion. It is important to note that the change will not be reflected in the physical storage file until it has been closed. If the application needs to replace messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Example

```
CMailMessage mailMessage;

// Compose a new message
mailMessage.Compose(lpszSender,
                            lpszRecipient,
                            NULL,
                            lpszSubject,
                            lpszMessage);

CMessageStore mailStorage;

if (mailStorage.OpenFile(lpszFileName, MIME_STORAGE_WRITE))
{
    LONG nLastMessage = 0;

    // Get the last message number in the message store
    mailStorage.GetMessageCount(&nLastMessage);
```

```
    if (nLastMessage > 0)
    {
        // Replace the last message in the message store
        if (!mailStorage.ReplaceMessage(nLastMessage, mailMessage))
        {
            // We were unable to replace the message
        }
    }

    // Close the message store
    mailStorage.CloseFile();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

FindMessage, GetMessage, DeleteMessage, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGESTORE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CMessageStore::StoreMessage Method

```
LONG StoreMessage(
   CMailMessage& mailMessage
);
```

```
LONG StoreMessage(
   HMESSAGE hMessage
);
```

The **StoreMessage** method stores the specified message in the current message store.

Parameters

hStorage

Handle to the message store.

mailMessage

A **CMailMessage** object which references the message that will be stored.

hMessage

Handle to the message that will be stored.

Return Value

If the method succeeds, the return value is the message number for the message that was just stored. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreMessage** method will always append the specified message to the storage file. If you want to replace a message in the message store, you should use the **DeleteMessage** method to mark the original message for deletion and then use **StoreMessage** to write the updated message to the message store.

Example

```
CMailMessage mailMessage;

// Compose a new message
mailMessage.Compose(lpszSender,
                            lpszRecipient,
                            NULL,
                            lpszSubject,
                            lpszMessage,
                            NULL,
                            MIME_CHARSET_DEFAULT,
                            MIME_ENCODING_DEFAULT);

CMessageStore mailStorage;

// Open the message storage file
if (mailStorage.OpenFile(lpszFileName, MIME_STORAGE_WRITE))
{
    // Store a copy of the message in the message store
    nMessageId = mailStorage.StoreMessage(mailMessage);

    if (nMessageId == MIME_ERROR)
```

```
    {
        // We were unable to store the message
    }

    // Close the message store
    mailStorage.CloseFile();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

DeleteMessage, FindMessage, GetMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol Class Library

Query a network time server for the current date and time.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CNetworkTime
File Name	CSTIMV12.DLL
Version	12.0.3024.1286
LibID	79C32362-35E2-4115-9F08-043C7F982BB8
Import Library	CSTIMV12.LIB
Dependencies	None
Standards	RFC 868, RFC 5905

Overview

The Network Time Protocol class provides an interface for obtaining the current date and time from a server. The time is returned in Coordinated Universal Time (UCT) and can be adjusted for the local host's timezone. The time value is returned as the number of seconds which have elapsed since midnight, January 1, 1900. The class supports the Network Time Protocol (NTP) as specified in RFC 5905 and the TIME protocol described in RFC 868.

It is recommended you update your applications to call the methods which use 64-bit integer time values and use the NTP v4 protocol, which is supported in the current version of this API. To use NTP v4, provide the name of a valid NTP server and specify port 123. Although the API continues to support the TIME protocol on port 37, its use has been deprecated and applications should be updated to use NTP which is considered a more reliable method for obtaining the current time over the Internet.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol Class Methods

Class	Description
CNetworkTime	Constructor which initializes the current instance of the class
~CNetworkTime	Destructor which releases resources allocated by the class
Method	Description
ConvertTime	Convert between network and system time values
DisableTrace	Disable logging of network function calls
EnableTrace	Enable logging of network function calls to a file
FormatTime	Return the specified network time as a string formatted for the current locale.
GetErrorString	Return a description for the specified error code
GetLastError	Return the last error code
GetTime	Return the current network time from the server
IsInitialized	Determine if the class has been successfully initialized
SetLastError	Set the last error code
SetTime	Set the local system time with the network time

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::~CNetworkTime

```
~CNetworkTime();
```

The **CNetworkTime** destructor releases resources allocated by the current instance of the **CNetworkTime** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CNetworkTime** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CNetworkTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::CNetworkTime Method

```
CNetworkTime();
```

The **CNetworkTime** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **TimeInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CNetworkTime, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::ConvertTime Method

```
BOOL ConvertTime(
   ULONGLONG ulNetworkTime,
   LPSYSTEMTIME lpSystemTime,
   BOOL bLocalTime
);

BOOL ConvertTime(
   ULONGLONG ulNetworkTime,
   TIME_T* lpUnixTime
);

BOOL ConvertTime(
   ULONGLONG ulNetworkTime,
   CTime& localTime
);

BOOL ConvertTime(
   LPSYSTEMTIME lpSystemTime,
   ULONGLONG* lpNetworkTime
);
```

The **ConvertTime** method converts between a 32-bit network time value and a SYSTEMTIME structure.

Parameters

ulNetworkTime

The network time to be converted.

lpUnixTime

A pointer to a long integer which will contain the time in UNIX format. The value is the number of seconds since 1 January 1970 UTC and is commonly used with the standard C library time functions.

localTime

A **CTime** object which will contain the local time when the method returns.

lpSystemTime

A pointer to a SYSTEMTIME structure which will be modified for the specified network time.

bLocalTime

A boolean flag that is used to specify if the network time should be adjusted for the local timezone.

Return Value

If the network time can be converted, the method returns a non-zero value. If the network time cannot be converted, or the pointer to the SYSTEMTIME structure is invalid, the method will return zero.

Remarks

The network time value is a 64-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatTime, GetTime, SetTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::FormatTime Method

```
INT FormatTime(
   ULONGLONG ulNetworkTime,
   LPTSTR lpszDateTime,
   INT nMaxLength,
   BOOL bLocalTime
);

INT FormatTime(
   ULONGLONG ulNetworkTime,
   CString& strDateTime,
   BOOL bLocalTime
);
```

The **FormatTime** method converts the network time to a string value which contains the date and time formatted for the current locale.

Parameters

ulNetworkTime

An unsigned integer value which specifies the network time. This value should specify the number of seconds since midnight, January 1, 1900 UTC.

lpszDateTime

A null-terminated string buffer which will contain the formatted date and time. This buffer must be large enough to store both date and time values, including the terminating null character. The date is formatted using the current system locale's short date format. The time is formatted to use a 24-hour clock and includes the seconds value. An alternate version of this method accepts a reference to a **CString** object which will contain the date and time when the method returns.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. If the buffer is not large enough to contain the entire date and time string, the function will fail and return a value of zero.

bLocalTime

An integer boolean value which specifies if the network time should be adjusted for the local timezone. If this parameter is zero, the date and time are formatted using Coordinated Universal Time (UTC). If this parameter is non-zero, the network time value will be adjusted for the current timezone and will return a localized time value. The current timezone is determined by the configuration of the local system.

Return Value

If successful, the method returns the number of characters copied into the string buffer, not including the terminating null character. If the method fails, it returns a value of zero. Call the **GetLastError** method for more information about the cause of the failure.

Remarks

The **FormatTime** method is a convenient way to convert a numeric network time value to a string value which can be displayed to the user. The actual format of the string is determined by the local system configuration and locale. For example, in North America the date string will be returned using a month/day/year format, while European locales may return the date in a day/month/year format.

Example

```
// Request the time from a server and write it to standard output
CNetworkTime timeClient;
ULONGLONG ulNetworkTime = timeClient.GetTime(_T("time.nist.gov"),
TIME_PORT_DEFAULT, TIME_TIMEOUT);

if (ulNetworkTime != 0)
{
    CString strDateTime;
    INT nLength = timeClient.FormatTime(ulNetworkTime, strDateTime, TRUE);

    if (nLength > 0)
        _tprintf(_T("The local date and time is %s\n"), (LPCTSTR)strDateTime);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertTime, GetTime, SetTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::GetTime Method

```
BOOL GetTime(
   LPCTSTR lpszHostName,
   UINT nHostPort,
   ULONGLONG* lpNetworkTime
   UINT nTimeout
);

BOOL GetTime(
   LPCTSTR lpszHostName,
   UINT nHostPort,
   LPSYSTEMTIME lpSystemTime
   UINT nTimeout
);

BOOL GetTime(
   LPCTSTR lpszHostName,
   UINT nHostPort,
   CTime& localTime
   UINT nTimeout
);
```

The **GetTime** method returns the network time from the specified host.

Parameters

lpszHostName

The name of the server. The host must be running a time server that complies with the specifications outlined in RFC 868.

nHostPort

The port the time server is running on. A value of zero indicates that the default port number for the service should be used. By default, port 123 is used to connect to the time server.

lpNetworkTime

A pointer to an unsigned 64-bit integer which will contain the current network time when the method returns. Note that this value is not the same as UNIX time and cannot be used with the standard C time functions.

lpSystemTime

A pointer to a SYSTEMTIME structure that will contain the system time when the method returns. This structure can be used with many of the standard Windows API time-related functions.

localTime

A **CTime** object that will contain the local time when the method returns.

nTimeout

The number of seconds that the method will wait for a response from the server.

Return Value

If the method succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the method was unable to obtain the time from the specified host, it returns zero.

Remarks

The **GetTime** method will cause the calling thread to block until the time is returned by the server, or

the operation times out. If you do not want the current thread to block, you can create a worker thread and call this function from within that thread. The blocking network call in the worker thread will not interefere with any other threads in your application.

In most cases, it is preferable to use the NTP pool servers which distribute requests over a number of different servers:

- **pool.ntp.org** is a global load-balanced pool of time servers
- **us.pool.ntp.org** servers are located in the United States
- **eu.pool.ntp.org** servers are located in Europe
- **asia.pool.ntp.org** servers are located in Asia

There are also load balanced pools of time servers available in specific countries prefixed by their country code. For example, the **de.pool.ntp.org** servers are located in Germany and the **uk.pool.ntp.org** servers are located in the United Kingdom. Regardless of their location, the time values they return are always in Coordinated Universal Time (UTC) and are not localized based on the physical location of the server.

In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a-g.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b-g.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a-b.nist.gov	132.163.96.1	Boulder, Colorado
time-b-b.nist.gov	132.163.96.2	Boulder, Colorado
time-c-b.nist.gov	132.163.96.3	Boulder, Colorado

Time servers are also commonly maintained by government agencies and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they support the standard time service available on port 123.

If you are requesting the time from a public server, you should ensure that your application does not send too many requests over a short period of time. In most cases, you should not query a time server more than once per minute. Most public time servers will block the IP addresse of a client which sends too many requests.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch. NTP servers return the time as the number of seconds since January 1, 1900.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTime, FormatTime, SetTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

CNetworkTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the error code for the current thread. This method is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

GetErrorString, GetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::SetTime Method

```
BOOL SetTime(
   ULONGLONG ulNetworkTime
);

BOOL SetTime(
   LPSYSTEMTIME lpSystemTime
);
```

The **SetTime** method sets the local system clock to the specified date and time.

Parameters

ulNetworkTime

The date and time the system clock should be set to, represented as the number of seconds since midnight, 1 January 1900.

lpSystemTime

A pointer to a SYSTEMTIME structure that specifies the date and time that the local clock should be set to.

Return Value

If the method is able to update the local time, it returns a non-zero value. If the specified time is invalid, or the user does not have the access rights to change the system clock, the method returns zero.

Remarks

The **SetTime** method updates the local system clock to the specified date and time. The user must have the appropriate privileges to change the system clock, otherwise an error will be returned. This typically requires the application to execute with elevated privileges. Because Windows systems are usually configured to automatically synchronize the system clock with a time server using the Windows Time service (W32Time), most applications should not use this function.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertTime, GetTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNetworkTime::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol Data Structures

- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol Class Library

Download and submit articles to a news server.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CNntpClient
File Name	CSNWSV12.DLL
Version	12.0.3024.1286
LibID	C11CFCC7-8D02-4CB5-B3D2-003BD1D65C40
Import Library	CSNWSV12.LIB
Dependencies	None
Standards	RFC 977, RFC 2980

Overview

The Network News Transfer Protocol (NNTP) is used with servers that provide news services. This is similar in functionality to bulletin boards or message boards, where topics are organized hierarchically into groups, called newsgroups. Users can browse and search for messages, called news articles, which have been posted by other users. On many servers, they can also post their own articles which can be read by others. The largest collection of public newsgroups available is called USENET, a world-wide distributed discussion system. In addition, there are a large number of smaller news servers. For example, Catalyst Development operates a news server which methods as a forum for technical questions and announcements.

The SocketTools library provides a comprehensive interface for accessing newsgroups, retrieving articles and posting new articles. In combination with the Mail Message library to process the news articles, SocketTools can be used to integrate newsgroup access with an existing email application, or you can implement your own full-featured newsgroup client.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol Class Methods

Class	Description
CNntpClient	Constructor which initializes the current instance of the class
~CNntpClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Authenticate	Authenticate the specified user on the news server
Cancel	Cancel the current blocking operation
CloseArticle	Close the article being posted to the current newsgroup
Command	Send a command to the server
Connect	Connect to the specified server
CreateArticle	Create a new article in the current newsgroup
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of network function calls to a file
FreezeEvents	Suspend asynchronous event processing
GetArticle	Copy the specified article to a local buffer
GetArticleHeaders	Return the contents of the specified article header
GetArticleMessageId	Return the message identifier for the specified article
GetArticleRange	Return the first and last article number for the current group
GetCurrentArticle	Return the current article number for the selected group
GetCurrentDate	Return the current date and time
GetErrorString	Return a description for the specified error code
GetFirstArticle	Return the first available article in the currently selected newsgroup
GetFirstGroup	Return the first available newsgroup from the server
GetHandle	Return the client handle used by this instance of the class
GetGroupName	Return the name of the currently selected newsgroup

GetGroupTitle	Return a description of the currently selected newsgroup
GetLastError	Return the last error code
GetMessageIdArticle	Return the article number for the specified message identifier
GetNextArticle	Return the next available article from the currently selected newsgroup
GetNextGroup	Return the next available newsgroup from the server
GetResultCode	Return the result code from the previous command
GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
ListArticles	Return a list of articles in the currently selected newsgroup
ListGroups	Return a list of newsgroups maintained by the server
NntpEventProc	Callback method that processes events generated by the client
OpenArticle	Open the specified article in the currently selected newsgroup
OpenNextArticle	Open the next available article
OpenPreviousArticle	Open the previous article
PostArticle	Post a new article to the news server
Read	Read data returned by the news server
RegisterEvent	Register an event callback function
Reset	Reset the client
SelectGroup	Select the specified newsgroup to retrieve articles from
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
StoreArticle	Store the specified article to a file on the local system
Write	Write data to the news server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::CNntpClient Method

```
CNntpClient();
```

The **CNntpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **NntpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CNntpClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::~CNntpClient

```
~CNntpClient();
```

The **CNntpClient** destructor releases resources allocated by the current instance of the **CNntpClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CNntpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CNntpClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Authenticate Method

```
INT Authenticate(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **Authenticate** method is used to authenticate access to the news server. Not all news servers require authentication by the client.

Parameters

lpszUserName

Pointer to a string which specifies the user name required for authentication on the news server.

lpszPassword

Pointer to a string which specifies the password required for authentication on the news server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should only be called if the server requires authentication. Two authentication methods, "original" and "simple" authentication, are recognized by the library.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, ListArticles, ListGroups

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::CloseArticle Method

```
INT CloseArticle();
```

The **CloseArticle** method closes the current article that has been opened or created.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If an article is being created, this method actually submits the article to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

CreateArticle, OpenArticle, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Command Method

```
INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter,
   BOOL bMultiLine
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the class.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

bMultiLine

An optional boolean argument used to specify if multiple lines of data will be returned by the server as the result of the command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker. If this argument is omitted, ***bMultiLine*** is FALSE.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is NN. For secure connections, the default port number is NN.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
NNTP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
NNTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
NNTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
NNTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
NNTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will

	allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
NNTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the NNTP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials, Disconnect, GetSecurityInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::CreateArticle Method

```
INT CreateArticle();
```

The **CreateArticle** method creates a new article in the current newsgroup.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the POST command to the news server. Not all servers permit clients to post articles. The client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

The **CloseArticle** method must be called once the contents of the article has been written to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

CloseArticle, GetCurrentDate, ListArticles, OpenArticle, OpenNextArticle, OpenPreviousArticle, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              NNTP_PORT_SECURE,
                              NNTP_TIMEOUT,
                              NNTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This

	event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetArticle Method

```
INT GetArticle(
   LONG nArticleId,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

INT GetArticle(
   LONG nArticleId,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

INT GetArticle(
   LONG nArticleId,
   CString& strBuffer
);

INT GetArticle(
   LPCTSTR lpszMessageId,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

INT GetArticle(
   LPCTSTR lpszMessageId,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

INT GetArticle(
   LPCTSTR lpszMessageId,
   CString& strBuffer
);
```

The **GetArticle** method retrieves the specified article and copies the contents to a local buffer.

Parameters

nArticleId

An integer value which specifies the number of the article to retrieve from the server. This value must be greater than zero. Overloaded versions of this method also support article IDs that are 64-bit integers.

lpszMessageId

A null-terminated string which specifies the message ID of the article to retrieve from the server. This parameter cannot be NULL or specify an empty string.

lpBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns,

this value will be updated with the actual length of the file that was downloaded.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetArticle** method is used to retrieve an article from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the article. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the article data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the complete article has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the article into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = pClient->GetArticle(nArticleId, &hgblBuffer, &cbBuffer);

if (nResult != NNTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // article text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

CreateArticle, EnableEvents, GetArticleHeaders, GetTransferStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetArticleHeaders Method

```
INT GetArticleHeaders(
   LONG nArticleId,
   LPBYTE lpHeaders,
   LPDWORD lpdwLength
);

INT GetArticleHeaders(
   LONG nArticleId,
   HGLOBAL *lpHeaders,
   LPDWORD lpdwLength
);

INT GetArticleHeaders(
   LONG nArticleId,
   CString& strHeaders
);
```

The **GetArticleHeaders** method retrieves the headers for the specified article from the server.

Parameters

nArticleId

Number of article to retrieve from the server. This value must be greater than zero. Overloaded versions of this method also support article IDs that are 64-bit integers.

lpHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpHeaders*** parameter. If the ***lpHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetArticleHeaders** method is used to retrieve an article header block from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpHeaders*** parameter point to a global memory handle which will contain the message headers when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the transfer completes, a timeout occurs or

the transfer is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetArticle, GetArticleRange, ListArticles, PostArticle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetArticleMessageId Method

```
INT GetArticleMessageId(
   LONG nArticleId,
   LPTSTR lpszMessageId,
   INT cbMessageId
);

INT GetArticleMessageId(
   LONG nArticleId,
   CString& strMessageId
);
```

The **GetArticleMessageId** method returns the message identifier for the specified article in the current newsgroup.

Parameters

nArticleId

Article number to retrieve the message identifier for. The value may be zero, in which case the current article number is used.

lpszMessageId

Pointer to a string buffer which will contain the message identifier for the specified article.

cbMessageId

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the message identifier string. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifier is a string which can uniquely identify the message on the news server. This value may be used to retrieve the contents of the article.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessageIdArticle, ListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetArticleRange Method

```
BOOL GetArticleRange(
   LONG *lpnFirstArticle,
   LONG *lpnLastArticle,
   LONG *lpnArticleCount
);

BOOL GetArticleRange(
   ULONGLONG *lpnFirstArticle,
   ULONGLONG *lpnLastArticle
   ULONGLONG *lpnArticleCount
);
```

The **GetArticleRange** method returns the first and last article numbers for the currently selected newsgroup.

Parameters

lpnFirstArticle

Pointer to an integer that will contain the first article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnLastArticle

Pointer to an integer that will contain the last article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnArticleCount

Pointer to an integer that will contain the total number of articles in currently selected newsgroup. If this parameter is NULL, it will be ignored.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

OpenArticle, ListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetCurrentArticle Method

```
BOOL GetCurrentArticle(
   LONG *lpnArticleId
);

BOOL GetCurrentArticle(
   ULONGLONG *lpnArticleId
);
```

The **GetCurrentArticle** method returns the current article number for the currently selected newsgroup.

Parameters

lpnArticleId

Pointer to an integer variable that will contain the current article number when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

OpenArticle, GetArticleRange, GetFirstArticle, GetNextArticle, ListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetCurrentDate Method

```
INT GetCurrentDate(
   LPTSTR lpszDate,
   INT nMaxLength
);

INT GetCurrentDate(
   CString& strDate
);
```

The **GetCurrentDate** method copies the current date and time to the specified buffer in a format that is commonly used in news articles. This date format should be used in all date-related fields in the message header.

Parameters

lpszDate

Pointer to a string buffer that will contain the current date and time when the method returns.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateArticle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetFirstArticle Method

```
BOOL GetFirstArticle(
   LPNEWSARTICLE lpArticle
);

BOOL GetFirstArticle(
   LPNEWSARTICLEEX lpArticle
);
```

The **GetFirstArticle** method returns information about the first article in the currently selected newsgroup.

Parameters

lpArticle

A pointer to a NEWSARTICLE or NEWSARTICLEEX structure which will contain information about the first article in the currently selected directory. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSARTICLEEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no articles in the current newsgroup, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstArticle** method returns information about the first article in the currently selected newsgroup. This method is used in conjunction with the **GetNextArticle** method to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside a loop calling **GetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NttpGetArticle** for each article number to retrieve the article text.

The date and time that the article was posted is returned in the *stPosted* member of the NEWSARTICLE structure. This value is returned in Universal Coordinated Time (UTC) and can be converted to local time using the **SystemTimeToTzSpecificLocalTime** function.

Example

```
// List all articles in the current group
if (pClient->ListArticles() == NNTP_ERROR)
    pClient->ShowError();
else
{
    NEWSARTICLE newsArticle;
    BOOL bResult;

    // Get each article in the current newsgroup, printing the article
    // number and the subject of the article
    bResult = pClient->GetFirstArticle(&newsArticle);
    while (bResult)
    {
        _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
        bResult = pClient->GetNextArticle(&newsArticle);
```

```
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetArticle, GetFirstGroup, GetNextArticle, GetNextGroup, ListArticles, ListGroups, SelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetFirstGroup Method

```
BOOL GetFirstGroup(
   LPNEWSGROUP lpGroup
);

BOOL GetFirstGroup(
   LPNEWSGROUPEX lpGroup
);
```

The **GetFirstGroup** method returns information about the first available newsgroup.

Parameters

lpGroup

A pointer to a NEWSGROUP or NEWSGROUPEX structure which will contain information about the first available newsgroup. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSGROUPEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no newsgroups available, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstGroup** method returns information about the first newsgroup on the server. This method is used in conjunction with the **GetNextGroup** method to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

Example

```
// List all available newsgroups
if (pClient->ListGroups() == NNTP_ERROR)
    pClient->ShowError();
else
{
    NEWSGROUP newsGroup;
    BOOL bResult;

    // Get each newsgroup, printing the article range and
    // the name of the group
    bResult = pClient->GetFirstGroup(&newsGroup);
    while (bResult)
    {
        printf("%ld %ld %s\n", newsGroup.nFirstArticle,
                               newsGroup.nLastArticle,
                               newsGroup.szName);
        bResult = pClient->GetNextGroup(&newsGroup);
    }
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstArticle, GetNextArticle, GetNextGroup, ListArticles, ListGroups, SelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetGroupName Method

```
INT GetGroupName(
   LPTSTR lpszGroupName,
   INT nMaxLength
);

INT GetGroupName(
   CString& strGroupName
);
```

The **GetGroupName** method returns the name of the currently selected newsgroup.

Parameters

lpszGroupName

Pointer to a string buffer that will contain the name of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the newsgroup name. If no newsgroup has been selected, the method will return a value of zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetGroupTitle, ListGroups, SelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetGroupTitle Method

```
INT GetGroupTitle(
   LPTSTR lpszGroupTitle,
   INT nMaxLength
);

INT GetGroupTitle(
   CString& strGroupTitle
);
```

The **GetGroupTitle** method returns a description of the currently selected newsgroup.

Parameters

lpszGroupTitle

Pointer to a string buffer that will contain a description of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the description. If no newsgroup has been selected, the method will return a value of zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The news server must support the XGTITLE command so that the group description can be obtained when the newsgroup is selected. If this command is not recognized, then no description will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetGroupName, ListGroups, SelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetMessageIdArticle Method

```
BOOL GetMessageIdArticle(
   LPCTSTR lpszMessageId,
   LONG *lpnMessageId
);

BOOL GetMessageIdArticle(
   LPCTSTR lpszMessageId,
   ULONGLONG *lpnMessageId
);
```

The **GetMessageIdArticle** method returns the article number associated with the specified message identifier in the current newsgroup.

Parameters

lpszMessageId

The message identifier string.

lpnArticleId

Pointer to an integer variable that will contain the article number associated with the message ID.

Return Value

If the method succeeds, the return value is the article number. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetArticleMessageId, ListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetNextArticle Method

```
BOOL GetNextArticle(
   LPNEWSARTICLE lpArticle
);

BOOL GetNextArticle(
   LPNEWSARTICLEEX lpArticle
);
```

The **GetNextArticle** method returns information about the next article in the currently selected newsgroup.

Parameters

lpArticle

A pointer to a NEWSARTICLE or NEWSARTICLEEX structure which will contain information about the next available article in the currently selected directory. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSARTICLEEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no more articles in the current newsgroup, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextArticle** method returns information about the next available article in the currently selected newsgroup. This method is used in conjunction with the **GetFirstArticle** method to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside a loop calling **GetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NttpGetArticle** for each article number to retrieve the article text.

Example

```
// List all articles in the current group
if (pClient->ListArticles() == NNTP_ERROR)
    pClient->ShowError();
else
{
    NEWSARTICLE newsArticle;
    BOOL bResult;

    // Get each article in the current newsgroup, printing the article
    // number and the subject of the article
    bResult = pClient->GetFirstArticle(&newsArticle);
    while (bResult)
    {
        _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
        bResult = pClient->GetNextArticle(&newsArticle);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetArticle, GetFirstArticle, GetFirstGroup, GetNextGroup, ListArticles, ListGroups, SelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetNextGroup Method

```
BOOL GetNextGroup(
   LPNEWSGROUP lpGroup
);

BOOL GetNextGroup(
   LPNEWSGROUPEX lpGroup
);
```

The **GetNextGroup** method returns information about the next available newsgroup.

Parameters

lpGroup

A pointer to a NEWSGROUP or NEWSGROUPEX structure which will contain information about the next available newsgroup. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSGROUPEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no more newsgroups available, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextGroup** method returns information about the next newsgroup on the server. This method is used in conjunction with the **GetFirstGroup** method to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
// List all available newsgroups
if (pClient->ListGroups() == NNTP_ERROR)
    pClient->ShowError();
else
{
    NEWSGROUP newsGroup;
    BOOL bResult;

    // Get each newsgroup, printing the article range and
    // the name of the group
    bResult = pClient->GetFirstGroup(&newsGroup);
    while (bResult)
    {
        printf("%ld %ld %s\n", newsGroup.nFirstArticle,
                               newsGroup.nLastArticle,
                               newsGroup.szName);
        bResult = pClient->GetNextGroup(&newsGroup);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstArticle, GetFirstGroup, GetNextArticle, ListArticles, ListGroups, SelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

Command, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetResultString Method

```
INT GetResultString(
   LPTSTR lpszResult,
   INT cbResult
);

INT GetResultString(
   CString& strResult
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

The buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
NNTP_STATUS_UNUSED	No connection has been established.
NNTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
NNTP_STATUS_CONNECT	The client is establishing a connection with the server.
NNTP_STATUS_READ	The client is reading data from the server.
NNTP_STATUS_WRITE	The client is writing data to the server.
NNTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::GetTransferStatus Method

```
INT GetTransferStatus(
   LPNNTPTRANSFERSTATUS lpStatus
);

INT GetTransferStatus(
   LPNNTPTRANSFERSTATUSEX lpStatus
);
```

The **GetTransferStatus** method returns information about the current data transfer in progress.

Parameters

lpStatus

A pointer to an NNTPTRANSFERSTATUS or NNTPTRANSFERSTATUSEX structure which contains information about the status of the current data transfer. If the server can potentially return very large article IDs, it is recommended that you use the **NNTPTRANSFERSTATUSEX** structure.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If no article is currently being retrieved or submitted to the server, this function will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

CNntpClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::ListArticles Method

```
INT ListArticles(
   LONG nFirstArticle,
   LONG nLastArticle
);

INT ListArticles(
   ULONGLONG nFirstArticle,
   ULONGLONG nLastArticle
);
```

The **ListArticles** method returns a list of articles in the currently selected newsgroup, within the specified article range.

Parameters

nFirstArticle

The first newsgroup article to be returned in the list. If this value is -1, the list will begin with the first available article in the newsgroup.

nLastArticle

The last newsgroup article to be returned in the list. If the value is -1, the list will end with the last available article in the newsgroup.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. Use the **GetFirstArticle** and **GetNextArticle** methods to read the list of articles returned by the server. If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Example

```
// List all articles in the current group
if (pClient->ListArticles() == NNTP_ERROR)
    pClient->ShowError();
else
{
    NEWSARTICLE newsArticle;
    BOOL bResult;

    // Get each article in the current newsgroup, printing the article
    // number and the subject of the article
    bResult = pClient->GetFirstArticle(&newsArticle);
    while (bResult)
    {
        printf("%ld %s %s\n", newsArticle.nArticleId, newsArticle.szSubject);
        bResult = pClient->GetNextArticle(&newsArticle);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetArticleRange, GetCurrentArticle, GetFirstArticle, GetNextArticle, ListGroups

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::ListGroups Method

```
INT ListGroups(
   LPCTSTR lpszLastUpdated,
   BOOL bLocalTime
);
```

The **ListGroups** method instructs the server to begin sending a list of newsgroups that were created since the specified date.

Parameters

lpszLastUpdated

Pointer to a string which specifies the date and time that the list of newsgroups were last retrieved from the server. This parameter may be NULL or an empty string, in which case all available newsgroups will be listed by the server.

bLocalTime

A boolean value which indicates if the time specified in the *lpszLastUpdated* parameter is for the current timezone. If the value is non-zero, the time is assumed to be in the local timezone. If the value is zero, the time is assumed to be in Coordinated Universal Time (UTC).

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ListGroups** method is used in conjunction with the **GetFirstGroup** and **GetNextGroup** methods to enumerate all of the newsgroups that were added to the server since a specific date and time. Typically this is used to provide the user with a list of updated newsgroups to select. To list all of the newsgroups available on the server, omit the arguments to this method.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
LPCTSTR lpszUpdated = _T("1/1/2004 12:00 AM");

// List all newsgroups that were added after a specific date
if (pClient->ListGroups(lpszUpdated, TRUE) == NNTP_ERROR)
    pClient->ShowError();
else
{
    NEWSGROUP newsGroup;
    BOOL bResult;

    // Get each newsgroup, printing the article range and
    // the name of the group
    bResult = pClient->GetFirstGroup(&newsGroup);
    while (bResult)
    {
        printf("%ld %ld %s\n", newsGroup.nFirstArticle,
```

```
                                newsGroup.nLastArticle,
                                newsGroup.szName);
            bResult = pClient->GetNextGroup(&newsGroup);
        }
    }
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetArticleRange, GetFirstGroup, GetNextGroup, ListGroups, SelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::OpenArticle Method

```
INT OpenArticle(
   LONG nArticleId
);

INT OpenArticle(
   LPCTSTR lpszMessageId
);
```

The **OpenArticle** method opens the specified article in the currently selected newsgroup.

Parameters

nArticleId

Number that specifies which article in the current newsgroup to retrieve. This value may be zero, which specifies that the current article should be returned.

lpszMessageId

Pointer to a string which contains the message identifier for the article in the current newsgroup.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

CloseArticle, GetArticle, GetArticleRange, GetCurrentArticle, ListArticles, OpenNextArticle, OpenPreviousArticle, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::OpenNextArticle Method

```
INT OpenNextArticle();
```

The **OpenNextArticle** method opens the next available article in the current newsgroup.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

CloseArticle, GetArticle, GetArticleRange, GetCurrentArticle, ListArticles, OpenPreviousArticle, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::OpenPreviousArticle Method

```
INT OpenPreviousArticle();
```

The **OpenPreviousArticle** method opens the previous article in the current newsgroup.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

CloseArticle, GetArticle, GetArticleRange, GetCurrentArticle, ListArticles, OpenNextArticle, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::PostArticle Method

```
INT PostArticle(
   LPCTSTR lpBuffer,
   DWORD dwLength
);
```

The **PostArticle** method post the contents of the specified buffer to the server as a new article in the current newsgroup.

Parameters

lpBuffer

A pointer to a character buffer which contains the article to be posted to the currently selected newsgroup.

dwLength

Specifies the length of the string which contains the article. If this parameter is -1, the actual length of the string is calculated by searching the buffer for a terminating null byte.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostArticle** method is used to post the contents of the specified buffer to the server as a new article in the current newsgroup. Not all newsgroups permit new articles to be posted, and some newsgroups may require that you email the article to a moderator for approval instead of posting directly to the group. It may be required that the client authenticate itself using the **Authenticate** method prior to posting the article.

A news article is similar to an email message in that it contains one or more header fields, followed by an empty line, followed by the body of the article. Each line of text should be terminated by a carriage return/linefeed sequence of characters. The Mail Message library can be used to compose a message if needed. Note that the article header must contain a header field named "Newsgroups" with a value that specifies the newsgroup or newsgroups the article is being posted to. If this header field is missing, the news server will reject the article.

This method will cause the current thread to block until the transfer has completed, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetTransferStatus, IsBlocking, IsWritable, Read, Reset, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   NNTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
NNTP_EVENT_CONNECT	The connection to the server has completed.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **NntpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **NntpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a news article is being uploaded or downloaded. The NNTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one NNTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, NntpEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client cannot be reset while it is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

Cancel, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::SelectGroup Method

```
INT SelectGroup(
   LPCTSTR lpszGroupName
);
```

The **SelectGroup** method selects the specified newsgroup from which articles will be retrieved.

Parameters

lpszGroupName

Pointer to a string which specifies the newsgroup to be selected. This value may be NULL, in which case the current newsgroup is unchanged, but the article count is updated.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method selects the newsgroup and obtains a description and the first and last article numbers for that group.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetGroupName, GetGroupTitle, ListGroups

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::StoreArticle Method

```
INT StoreArticle(
   LONG nArticleId,
   LPCTSTR lpszFileName
);
```

The **StoreMessage** method retrieves an article from the current newsgroup and stores it in a local file.

Parameters

nArticleId

Number of the article to retrieve. This value must be greater than zero.

lpszFileName

Pointer to a string which specifies the file that the article will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the article.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreArticle** method provides a method of retrieving and storing an article on the local system. The contents of the article is stored as a text file, using the specified file name. This method always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

If event handling is enabled, the NNTP_EVENT_PROGRESS event will fire periodically during the transfer of the article to the local system. An application can determine how much of the article has been retrieved by calling the **GetTransferStatus** method.

To retrieve the message into a global memory buffer so that it can be passed to the MIME library, use the **GetArticle** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetArticle, GetArticleHeaders, GetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNntpClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol Data Structures

- NEWSARTICLE
- NEWSGROUP
- NNTPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSARTICLE

This structure is used by the **GetFirstArticle** and **GetNextArticle** methods to return information about news articles.

```
typedef struct _NEWSARTICLE
{
    LONG        nArticleId;
    LONG        nBytes;
    LONG        nLines;
    TCHAR       szSubject[NNTP_MAXSUBJLEN];
    TCHAR       szAuthor[NNTP_MAXAUTHLEN];
    TCHAR       szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR       szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME  stPosted;
} NEWSARTICLE, *LPNEWSARTICLE;
```

Members

nArticleId

A long integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A null-terminated string which specifies the subject of the article.

szAuthor

A null-terminated string which specifies the email address of the user who posted the article.

szMessageId

A null-terminated string which specifies the message ID for the article.

szReferences

A null-terminated string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Remarks

If the server uses 64-bit article IDs your application should use the NEWSARTICLEEX structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstArticle, GetNextArticle, ListArticles, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSARTICLEEX

This structure is used by the **GetFirstArticle** and **GetNextArticle** methods to return information about news articles.

```
typedef struct _NEWSARTICLEEX
{
    ULONGLONG   nArticleId;
    ULONG       nBytes;
    ULONG       nLines;
    TCHAR       szSubject[NNTP_MAXSUBJLEN];
    TCHAR       szAuthor[NNTP_MAXAUTHLEN];
    TCHAR       szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR       szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME  stPosted;
} NEWSARTICLEEX, *LPNEWSARTICLEEX;
```

Members

nArticleId

An unsigned 64-bit integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A null-terminated string which specifies the subject of the article.

szAuthor

A null-terminated string which specifies the email address of the user who posted the article.

szMessageId

A null-terminated string which specifies the message ID for the article.

szReferences

A null-terminated string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstArticle, GetNextArticle, ListArticles, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSGROUP

This structure is used by the **GetFirstGroup** and **GetNextGroup** methods to return information about the available newsgroups.

```
typedef struct _NEWSGROUP
{
    LONG        nFirstArticle;
    LONG        nLastArticle;
    DWORD       dwAccess;
    TCHAR       szName[NNTP_MAXGRPNAMLEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

A long integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

A long integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Value	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A null-terminated string which specifies the name of the newsgroup.

Remarks

If the server uses 64-bit article IDs your application should use the NEWSGROUPEX structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstGroup, GetNextGroup, NEWSARTICLE, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSGROUPEX

This structure is used by the **GetFirstGroup** and **GetNextGroup** methods to return information about the available newsgroups.

```
typedef struct _NEWSGROUPEX
{
    ULONGLONG   nFirstArticle;
    ULONGLONG   nLastArticle;
    DWORD       dwAccess;
    TCHAR       szName[NNTP_MAXGRPNAMLEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

An unsigned 64-bit integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

An unsigned 64-bit integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Value	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A null-terminated string which specifies the name of the newsgroup.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstGroup, GetNextGroup, NEWSARTICLE, NEWSARTICLEEX, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NNTPTRANSFERSTATUS

This structure is used by the GetTransferStatus method to return information about an article transfer in progress.

```
typedef struct _NNTPTRANSFERSTATUS
{
   UINT    nArticleId;
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} NNTPTRANSFERSTATUS, *LPNNTPTRANSFERSTATUS;
```

Members

nArticleId

The article ID of the current article that is being transferred. If an article is being posted, this member will be set to zero.

dwBytesTotal

The total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

GetTransferStatus, RegisterEvent, NNTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NNTPTRANSFERSTATUSEX

This structure is used by the **GetTransferStatus** method to return information about an article transfer in progress. This structure is intended for use with servers that return 64-bit article IDs.

```
typedef struct _NNTPTRANSFERSTATUSEX
{
   ULONGLONG nArticleId;
   DWORD     dwBytesTotal;
   DWORD     dwBytesCopied;
   DWORD     dwBytesPerSecond;
   DWORD     dwTimeElapsed;
   DWORD     dwTimeEstimated;
} NNTPTRANSFERSTATUSEX, *LPNNTPTRANSFERSTATUSEX;
```

Members

nArticleId

An unsigned 64-bit integer that specifies the article ID of the current article that is being transferred. If an article is being posted, this value will be zero.

dwBytesTotal

An unsigned integer that specifies the total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

An unsigned integer that specifies the total number of bytes that have been copied.

dwBytesPerSecond

An unsigned integer that specifies the average number of bytes that have been copied per second.

dwTimeElapsed

An unsigned integer that specifies the number of seconds that have elapsed since the transfer started.

dwTimeEstimated

An unsigned integer that specifies the estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

GetTransferStatus, RegisterEvent, NNTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Class Library

Retrieve and process the contents of a syndicated news feed.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CNewsFeed
File Name	CSRSSV12.DLL
Version	12.0.3024.1286
LibID	B8830D80-2618-4202-9C9C-9C3AD511E9D0
Import Library	CSRSSV12.LIB
Dependencies	None

Overview

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

News feeds can be accessed remotely from a web server, or locally as an XML formatted text file. The source of the feed is determined by the URI scheme that is specified. If the http or https scheme is specified, then the feed is retrieved from a web server. If the file scheme is used, the feed is considered to be local and is accessed from the disk or local network. The CNewsFeed class provides an interface that enables you to open a feed by URL and iterate through each of the items in the feed or search for a specific feed item. The class also provides a method that can be used to parse a string that contains XML data in RSS format, where the feed may have been retrieved from other sources such as a database.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must

have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Class Methods

Class	Description
CNewsFeed	Constructor which initializes the current instance of the class
~CNewsFeed	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
CloseFeed	Close the specified news feed and release memory allocated for the channel
DetachHandle	Detach the handle for the current instance of this class
DisableTrace	Disable logging of network function calls
EnableTrace	Enable logging of network function calls to a text file
FindItem	Find a specific news feed item using its unique identifier (GUID) property
GetErrorString	Return a description for the specified error code
GetFirstItem	Return information about the first item in the news feed channel
GetHandle	Return the client handle used by this instance of the class
GetItem	Return information about the specified news feed item
GetItemCount	Return the number of news feed items in the channel
GetItemProperty	Return the value of the specified news feed item property or attribute
GetItemText	Return the text description of the specified news feed item
GetLastError	Return the last error code
GetNextItem	Return information about the next item in the news feed channel
IsInitialized	Determine if the class has been successfully initialized
OpenFeed	Open the specified news feed and return information about the channel
ParseFeed	Parse the contents of a string and return information about the channel
RefreshFeed	Refresh the specified news feed, updating the items in the channel
SetLastError	Set the last error code
ShowError	Display a message box with a description of the specified error
StoreFeed	Store the contents of the specified news feed in an XML formatted text file
ValidateFeed	Validate the contents of the specified news feed, returning the number of items in the feed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::CNewsFeed Method

```
CNewsFeed();
```

The **CNewsFeed** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **RssInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CNewsFeed, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::~CNewsFeed

```
~CNewsFeed();
```

The **CNewsFeed** destructor releases resources allocated by the current instance of the **CNewsFeed** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CNewsFeed** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

CNewsFeed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::AttachHandle Method

```
VOID AttachHandle(
   HCHANNEL hChannel
);
```

The **AttachHandle** method attaches the specified handle to the current instance of the class.

Parameters

hChannel

The handle to the channel that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that news feed channel.

If a handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous news feed channel from being closed, you must call the **DetachHandle** method. Failure to close the detached handle may result in a resource leak in your application.

Note that the ***hChannel*** parameter is presumed to be valid and no checks are performed to ensure its validity. Specifying an invalid handle value will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::CloseFeed Method

```
BOOL CloseFeed();
```

The **CloseFeed** method closes the specified news feed and releases memory allocated for the channel.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CloseFeed** method must be called whenever the application has completed processing the news feed. It is important to note that the memory allocated for the channel will be released when this function is called, which means that any data referenced in the RSSCHANNEL and RSSCHANNELITEM structures will no longer be valid and must not be used by the application after the feed has been closed.

This method can fail if the feed is currently being updated, such as when the **RefreshFeed** method is called. In this case, the channel handle will not be released and the application must attempt to close the feed at a later time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

OpenFeed ParseFeed StoreFeed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::DetachHandle Method

```
HCHANNEL DetachHandle();
```

The **DetachHandle** method detaches the handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the handle associated with the current instance of the class object. If there is no open news feed, the value INVALID_CHANNEL will be returned.

Remarks

This method is used to detach a handle created by the class for use with the SocketTools API. Once the handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly closed at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::FindItem Method

```
BOOL FindItem(
   LPCTSTR lpszValue,
   DWORD dwOptions,
   LPRSSCHANNELITEM lpItem
);

BOOL FindItem(
   LPCTSTR lpszValue,
   LPRSSCHANNELITEM lpItem
);
```

The **FindItem** method searches for an item in the news feed channel which matches the unique identifier (GUID) value and returns information about that item.

Parameters

lpszValue

A null-terminated string which specifies the value of the item being searched for. This value should uniquely identify the item in the feed, and this parameter cannot be an empty string or NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSS_FIND_GUID	Search the feed for items with a matching GUID property value. This is the default option, and is the only item property that is guaranteed to be unique in the feed. The search is case-sensitive, requiring that the ***lpszValue*** parameter match the property value exactly.
RSS_FIND_LINK	Search the feed for items with a matching link property value. For feeds that do not specify a GUID property, this is the recommended option for searching for an item. The search is not case-sensitive.
RSS_FIND_TITLE	Search the feed for items with a matching title. This option should not be used if you must ensure that the item returned is unique in the feed because there may be multiple items with the same title in the feed. The search is not case-sensitive.
RSS_FIND_PUBDATE	Search the feed for items with a matching publishing date. This option should not be used if you must ensure that the item returned is unique in the feed because more than one item may have the same publishing date. The format of the date string must match the standard format used with the RSS protocol and the match is not case-sensitive.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

It is recommended that you use the RSS_FIND_GUID option with news feeds that are using version 2.0 or later of the RSS specification. If the feed uses an earlier version, items may not include a GUID property. It is also possible that a feed may omit the GUID property even though it is considered a requirement for the current RSS specification. For the broadest compatibility with all news feeds, an application should not depend on being able to search for a specific news feed item by its GUID.

Only the GUID property is guaranteed to be unique in the feed. If the feed does not specify GUIDs for the news items, the application must use an alternate criteria such as the item hyperlink or publishing date. If there are multiple items that match the ***lpszValue*** value, the first matching item will be returned.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstItem, GetItem, GetItemProperty, GetItemText, GetNextItem, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetFirstItem Method

```
BOOL GetFirstItem(
   LPRSSCHANNELITEM lpItem
);
```

The **GetFirstItem** method returns information about the first item in the news feed channel.

Parameters

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the first item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, it returns a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstItem** method is used in conjunction with the **GetNextItem** method to enumerate the available items in the specified news feed channel. If this method fails, it typically indicates that the channel does not contain any valid news items or that the format of the news feed is invalid.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindItem, GetItem, GetItemProperty, GetItemText, GetNextItem, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetHandle Method

```
HCHANNEL GetHandle();
```

The **GetHandle** method returns the handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the handle associated with the current instance of the class object. If there is no open news feed, the value INVALID_CHANNEL will be returned.

Remarks

This method is used to obtain the handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetItem Method

```
BOOL GetItem(
   UINT nItemId,
   LPRSSCHANNELITEM lpItem
);
```

The **GetItem** method returns information about the specified item in the news feed channel.

Parameters

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this method will fail.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, it returns a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetItem** method is used to return information about a specific item in the news feed. If this method fails, it typically indicates that the item ID is invalid or that the feed does not contain any valid news items. The **GetItemCount** method can be used to determine the number of items contained in the feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindItem, GetFirstItem, GetItemCount, GetItemProperty, GetItemText, GetNextItem, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetItemCount Method

```
INT GetItemCount();
```

The **GetItemCount** method returns the number of items in the news feed channel.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of items in the news feed. A value of zero indicates that the feed channel is empty. If the method fails, the return value is RSS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetItemCount** method is used to determine the number of items that are contained in the news feed channel, and therefore determine the maximum value of the item identifier which can be used to reference a specific item in the feed. This value is the same as the value specified by the *nItemCount* member of the **RSSCHANNEL** structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

FindItem, GetItem, GetItemProperty, GetItemText, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetItemProperty Method

```
INT GetItemProperty(
   UINT nItemId,
   LPCTSTR lpszProperty,
   LPCTSTR lpszAttribute,
   LPTSTR lpszValue,
   INT nMaxLength
);

INT GetItemProperty(
   UINT nItemId,
   LPCTSTR lpszProperty,
   LPCTSTR lpszAttribute,
   CString& strValue
);
```

The **GetItemProperty** method is used to return the value of a property for the specified item in the news feed channel.

Parameters

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this method will fail.

lpszProperty

A null-terminated string which specifies the name of the item property. This parameter cannot point to an empty string or specify a NULL pointer.

lpszAttribute

A null-terminated string which specifies the name of an attribute for the property. If this parameter is an empty string or NULL pointer, the method will return the value of the property, rather than an attribute of the specified property.

lpszValue

A null-terminated string buffer which will contain the value of the specified item property or attribute. This string should be large enough to contain the property value. If this parameter is a NULL pointer, it will be ignored and the method will only return the length of value for the specified property.

nMaxLength

The maximum number of characters that may be copied into the property value buffer. If the value of this parameter is zero, then the ***lpszValue*** parameter is ignored and the method will only return the length of the value for the specified property.

Return Value

If the method succeeds, the return value is the length of the property value string. A return value of zero indicates that the property does not contain any value. If the method fails, the return value is RSS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetItemProperty** method is primarily used with custom item properties that may be used with

extensions to the news feed. The standard properties for an item such as the title, link and description can be obtained using **GetItem** and related methods. However, if items in the feed contain custom properties that are not part of the standard RSS format, this method can be used to obtain those values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindItem, GetItem, GetItemText, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetItemText Method

```
INT GetItemText(
   UINT nItemId,
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT GetItemText(
   UINT nItemId,
   CString& strBuffer
);
```

The **GetItemText** method is used to return a copy of an item's description.

Parameters

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this method will fail.

lpszBuffer

A null-terminated string buffer which will contain the value of the item description. If this parameter is a NULL pointer, it will be ignored and the method will only return the length of the item description.

nMaxLength

The maximum number of characters that may be copied into the description buffer. If the value of this parameter is zero, then the ***lpszValue*** parameter is ignored and the method will only return the length of the item description.

Return Value

If the method succeeds, the return value is the length of the item description. A return value of zero indicates that the item does not have a description. If the method fails, the return value is RSS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetItemText** function is used to obtain a copy of the string that describes the specified item. Typically this is text that provides a summary of the news feed item and is used in conjunction with the item's title and hyperlink to additional content.

The content of an item description is typically either plain text or HTML formatted text. It is the responsibility of the application to display the content in a format is appropriate for the end-user.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindItem GetItem, GetItemProperty, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::GetNextItem Method

```
BOOL GetNextItem(
   LPRSSCHANNELITEM lpItem
);
```

The **GetNextItem** method returns information about the next item in the news feed channel.

Parameters

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the next item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, it returns a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextItem** method is used in conjunction with the **GetFirstItem** method to enumerate the available items in the specified news feed channel. If this method fails, it typically indicates that there are no more items in the news feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FindItem, GetFirstItem, GetItem, GetItemProperty, GetItemText, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

CNewsFeed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::OpenFeed Method

```
BOOL OpenFeed(
   LPCTSTR lpszFeedUrl,
   UINT nTimeout,
   DWORD dwOptions,
   LPRSSCHANNEL lpChannel
);

BOOL OpenFeed(
   LPCTSTR lpszFeedUrl,
   UINT nTimeout,
   LPRSSCHANNEL lpChannel
);

BOOL OpenFeed(
   LPCTSTR lpszFeedUrl,
   LPRSSCHANNEL lpChannel
);
```

The **OpenFeed** method is used to open a news feed and return a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedUrl

A null-terminated string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSS_OPTION_NONE	No additional options are specified and the news feed is processed using relaxed rules when checking the validity of the feed. The library will attempt to automatically compensate for a feed that is malformed or does not strictly conform to the RSS standard.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the method. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A news feed may be local or remote, depending on the URL that is specified. If a local file name or file URL is specified for the feed, then it is opened locally and no network access is required. If an http or https URL is specified, then **OpenFeed** will attempt to download the feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs.

Although the **OpenFeed** method will meet the needs of most applications, if you require more complex functionality such as retrieving the feed asynchronously in the background or event notifications for large transfers, you can use the SocketTools Hypertext Transfer Protocol API to download the news feed and then use the **ParseFeed** method to parse the contents.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFeed, ParseFeed, RefreshFeed, StoreFeed, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::ParseFeed Method

```
BOOL ParseFeed(
   LPCTSTR lpszFeedXml,
   DWORD dwOptions,
   LPRSSCHANNEL lpChannel
);

BOOL ParseFeed(
   LPCTSTR lpszFeedXml,
   LPRSSCHANNEL lpChannel
);
```

The **ParseFeed** method is used to parse the contents of a news feed, returning a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedXml

A null-terminated string which contains the contents of the news feed. The string must contain XML formatted data that conforms to the RSS standard specification. This parameter cannot specify an empty string or a NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSS_OPTION_NONE	No additional options are specified.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the method. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ParseFeed** method is an alternative to the **OpenFeed** method, enabling the application to process a news feed from alternative sources such as a database or compressed file. It is important to note that the string which contains the news feed XML must be properly formatted and conform to the RSS standard specification.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed,

the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFeed, OpenFeed, RefreshFeed, StoreFeed, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::RefreshFeed Method

```
BOOL RefreshFeed(
   LPRSSCHANNEL lpChannel
);
```

The **RefreshFeed** method reloads the news feed and updates the items in the channel.

Parameters

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the method. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **RefreshFeed** method is called, the news feed is reloaded from the original source and the items in the channel are updated. For news feeds that are frequently updated, the *nTimeToLive* member of the **RSSCHANNEL** structure can provide a hint to the application as to how frequently the feed should be refreshed.

If the news feed was originally opened using an http or https URL, this method will download the updated feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs. The same timeout period and options will be used as when the feed was originally opened.

The **RefreshFeed** method should only be used if the feed was opened using the **OpenFeed** method, otherwise the method will fail with an error indicating that the operation is not supported.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. The members of this structure returned by previous calls to either the **OpenFeed** or **RefreshFeed** methods will no longer be valid and should not be referenced. Likewise, the members of an **RSSCHANNELITEM** structure will no longer be valid after this method returns.

It is important that the application does not make any assumptions about the number of news items in the channel, or the content associated with those items after the **RefreshFeed** function has been called. For example, never assume that the number of items in the channel remains the same, or that the item IDs for each item remains the same. If you need to find a specific item in the news feed, use the **FindItem** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseFeed, FindItem, GetItem, OpenFeed, StoreFeed, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::StoreFeed Method

```
BOOL StoreFeed(
   LPCTSTR lpszFileName
);
```

The **StoreFeed** method stores the contents of the news feed in an XML formatted text file.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file on the local system. The contents of the news feed will be stored in this file. If the file does not exist, it will be created; otherwise it will overwrite the contents of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OpenFeed, ParseFeed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CNewsFeed::ValidateFeed Method

```
INT ValidateFeed(
   LPCTSTR lpszFeedUrl,
   UINT nTimeout,
   DWORD dwOptions,
   LPSYSTEMTIME lpstModified
);

INT ValidateFeed(
   LPCTSTR lpszFeedUrl,
   LPSYSTEMTIME lpstModified
);

INT ValidateFeed(
   LPCTSTR lpszFeedUrl
);
```

The **RssValidateFeed** method is used to validate a news feed, returning the number of items in the feed and the date it was last modified.

Parameters

lpszFeedUrl

A null-terminated string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is reserved for future use an should always have a value of zero.

lpstModified

A pointer to a **SYSTEMTIME** structure which will specify the date that the feed was last modified when the method returns. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is the number of items in the news feed channel. If the method fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssValidateFeed** method can be used to check that a news feed exists and is properly formatted. If the contents of the feed are valid, the method will return the number of items in the feed and the date that it was last modified. This can be useful for applications that want to periodically check a news feed and determine if the contents have changed.

The **SYSTEMTIME** structure that is populated by the method specifies the date when the feed was last modified. The method first checks the value of the lastBuildDate property of the feed channel. If that property is not defined, then it will use the value of the pubDate property. If neither are defined, then the structure members will have a value of zero.

The validation process imposes strict checks on the structure of the news feed and requires that it conform to the RSS specification. For example, the feed must have a title, link and description. Each item in the feed must have either a title or description, and the hyperlinks specified in the feed must be valid. If the feed XML is malformed, or a required property of the feed is invalid or missing, this method will fail.

If a news feed cannot be validated, it still may be possible to open the feed using the **OpenFeed** method. By default, relaxed rules are used when parsing the contents of the feed and it does not check to ensure all required properties are defined and have valid values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssOpenFeed, RssParseFeed, RssRefreshFeed, RssStoreFeed, SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Data Structures

- RSSCHANNEL
- RSSCHANNELITEM
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RSSCHANNEL

The **RSSCHANNEL** structure contains information about the properties of the news feed channel.

```
typedef struct _RSSCHANNEL
{
    UINT        nItemCount;
    UINT        nTimeToLive;
    WORD        wVersionMajor;
    WORD        wVersionMinor;
    DWORD       dwFlags;
    DWORD       dwReserved;
    LPCTSTR     lpszTitle;
    LPCTSTR     lpszLink;
    LPCTSTR     lpszDescription;
    LPCTSTR     lpszCategory;
    LPCTSTR     lpszLanguage;
    LPCTSTR     lpszCopyright;
    LPCTSTR     lpszEditor;
    LPCTSTR     lpszWebmaster;
    LPCTSTR     lpszGenerator;
    LPCTSTR     lpszImageLink;
    LPCTSTR     lpszImageTitle;
    LPCTSTR     lpszImageUrl;
    SYSTEMTIME  stPublished;
    SYSTEMTIME  stLastBuild;
}  RSSCHANNEL, *LPRSSCHANNEL;
```

Members

nItemCount

An integer value which specifies number of news items in the channel.

nTimeToLive

An integer value which specifies the frequency in seconds at which the feed should be refreshed to obtain updated information. Not all feeds specify a time-to-live, in which case this member will have a value of zero.

wVersionMajor

A word value which specifies the major version number for the news feed.

wVersionMinor

A word value which specifies the minor version number for the news feed.

dwFlags

A value which specifies one or more option flags for the news feed channel. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A null-terminated string which specifies the name of the channel. If the content of the news feed corresponds to a website, this is typically the same as the title of the website. If a title has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a title.

lpszLink

A null-terminated string which specifies a URL to the website corresponding to the channel. Note

that this is not the URL of the news feed itself. Typically it is a link to the home page of the site which owns the news feed. If a link has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a valid link URL.

lpszDescription

A null-terminated string which describes the channel. This provides an overview of the news feed and the type of information that is provided. If a description of the feed has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a description.

lpszCategory

A null-terminated string which defines the category or categories that the channel belongs to. This property is optional and the category names themselves are user-defined. If a category has not been specified, this member will be NULL.

lpszLanguage

A null-terminated string which defines the language the channel is written in, using the standard language codes. This property is optional and if this member is NULL, the English language is typically presumed to be the default.

lpszCopyright

A null-terminated string which specifies a copyright notice for the content. If a copyright has not been specified, this member will be NULL.

lpszEditor

A null-terminated string which identifies the person responsible for managing the content of the news feed. If this property is defined, it is typically the name and email address of the feed editor. If an editor has not been specified, this member will be NULL.

lpszWebmaster

A null-terminated string which identifies the person responsible for technical issues related to the news feed. If this property is defined, it is typically the name and email address of a system administrator. If a webmaster has not been specified, this member will be NULL.

lpszGenerator

A null-terminated string which identifies the application that was used to create the news feed. If the application that generated the feed has not been specified, this member will be NULL.

lpszImageLink

A null-terminated string which specifies a URL to the website corresponding to the channel. In most cases, this is the same URL that is specified by the ***lpszLink*** member. If an image link has not been specified, this member will be NULL.

lpszImageTitle

A null-terminated string which identifies the image associated with the channel. This is usually a brief description of the image, and may be the same as the value specified by the ***lpszTitle*** member. If an image title has not been specified, this member will be NULL.

lpszImageUrl

A null-terminated string which specifies a URL for the image associated with the channel. An application can download this image and display it with the contents of the news feed. If an image URL has not been specified, this member will be NULL.

stPublished

The date that the news feed was published. For example, a feed that is associated with a weekly print publication may update this value once per week. Note that this is not necessarily the date that

the feed was last modified. If the channel does not specify the publish date, this structure will contain all zeroes.

stLastBuild

The date that the content of the channel was last modified. If the channel does not specify the build date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RSSCHANNELITEM

The **RSSCHANNELITEM** structure contains information about the properties of an item in a news feed channel.

```
typedef struct _RSSCHANNELITEM
{
    UINT        nItemId;
    DWORD       dwFlags;
    DWORD       dwReserved;
    LPCTSTR     lpszTitle;
    LPCTSTR     lpszLink;
    LPCTSTR     lpszText;
    LPCTSTR     lpszGuid;
    LPCTSTR     lpszAuthor;
    LPCTSTR     lpszSource;
    LPCTSTR     lpszComments;
    LPCTSTR     lpszEnclosure;
    SYSTEMTIME  stPublished;
}  RSSCHANNELITEM, *LPRSSCHANNELITEM;
```

Members

nItemId

An integer which identifies this item in the channel.

dwFlags

A value which specifies one or more option flags for the item. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A null-terminated string which specifies the title of the item. If a title for the item has not been specified, this member will be NULL.

lpszLink

A null-terminated string which specifies a URL that typically links to additional information related to the item. If a link for the item has not been specified, this member will be NULL.

lpszText

A null-terminated string which specifies a summary or description of the item. This may contain either plain text or HTML formatted text, and there is no fixed limit to the length of the text. If no text has been specified for the item, this member will be NULL.

lpszGuid

A null-terminated string which uniquely identifies the item in the channel. If this property is defined, it is guaranteed to be a unique, persistent value. It is important to note that this string does not have to be a standard GUID reference number, it can be any unique string. In many cases it is the same value as the item hyperlink specified by the ***lpszLink*** member, although an application should never depend on this behavior. If there is no unique identifier associated with the item, this member will be NULL.

lpszAuthor

A null-terminated string which identifies the author of the item. If this property is defined, it is

typically the name and email address of the person who created the content that the item links to. If the author is not specified, this member will be NULL.

lpszSource

A null-terminated string which identifies the source of the item, specified as a URL for the original news feed that contained it. This typically used to propagate credit for items that are aggregated by a third-party and re-published in their own channel. If the source is not specified, this member will be NULL.

lpszComments

A null-terminated string which specifies a URL that links to further discussion about the item. Typically this is a link to the comment area of a weblog or a forum topic specific to the item. If a comment link is not specified, this member will be NULL.

lpszEnclosure

A null-terminated string which specifies a URL that links to a file related to the item. This is similar to an attachment in an email message, however instead of the item containing the contents of the attached file, it only specifies a link to the file. Enclosures are most commonly used with podcasting where an item is linked to an audio or video file, however the link may reference any type of file. If there is no enclosure specified for the item, this member will be NULL.

stPublished

The date that the item was published. If the item does not specify the publish date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol Class Library

List and retrieve email messages from a mail server.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CPopClient
File Name	CSPOPV12.DLL
Version	12.0.3024.1286
LibID	33EB8402-9A5F-4ADD-B520-9C24EC56F1A1
Import Library	CSPOPV12.LIB
Dependencies	None
Standards	RFC 1939

Overview

The Post Office Protocol (POP3) provides access to a user's new email messages on a mail server. Functions are provided for listing available messages and then retrieving those messages, storing them either in files or in memory. Once a user's messages have been downloaded to the local system, they are typically removed from the server. The class provides a complete interface for managing a user's mailbox and is typically used in conjunction with the CMailMessage class, which is used to parse the messages retrieved from the server.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing

applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol Class Methods

Class	Description
CPopClient	Constructor which initializes the current instance of the class
~CPopClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
ChangePassword	Change the specified mail account password
Command	Send a command to the server
Connect	Connect to the specified server
CreateSecurityCredentials	Allocate a structure to establish client security credentials
DeleteMessage	Delete the specified message from the mailbox
DeleteSecurityCredentials	Delete the specified client security credentials
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable the event notification mechanism
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable the client event notification mechanism
EnableTrace	Enable logging of network function calls to a file
FreezeEvents	Suspend and resume event handling by the client
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetHeaderValue	Return the value of the specified header field
GetLastError	Return the last error code
GetMessage	Retrieve the specified message from the server
GetMessageCount	Return the number of messages available in the mailbox
GetMessageHeaders	Retrieve the specified message header from the server
GetMessageId	Return the message ID string for the specified message
GetMessageSender	Return the address of the message sender
GetMessageSize	Return the size of the specified message
GetMessageUID	Return the unique identifier for the specified message
GetResultCode	Return the result code from the previous command

GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current status of the client
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
Login	Login to the server
OpenMessage	Open the specified message for reading on the server
PopEventProc	Callback method that processes events generated by the client
Read	Read data returned by the server
RegisterEvent	Register an event handler for the specified event
Reset	Reset the client and return to a command state
SendMessage	Send a message through the mail server
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
StoreMessage	Store the contents of a message in the specified file
Write	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::CPopClient Method

```
CPopClient();
```

The **CPopClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **PopInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CPopClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::~CPopClient

```
~CPopClient();
```

The **CPopClient** destructor releases resources allocated by the current instance of the **CPopClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CPopClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CPopClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::ChangePassword Method

```
BOOL ChangePassword(
   LPCTSTR lpszUserName,
   LPCTSTR lpszOldPassword,
   LPCTSTR lpszNewPassword
);
```

The **ChangePassword** method changes the account password for the specified user.

Parameters

lpszUserName

Pointer to a string which specifies the user name of the account who's password will be changed. It is not required that this be the same user name that was used to login to the mail server.

lpszOldPassword

Pointer to a string which specifies the current account password.

lpszNewPassword

Pointer to a string which specifies the new account password. When the method returns, the user's mailbox password will be set to this value.

Return Value

If the method succeeds, it will return a non-zero value. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

The **ChangePassword** method is used to change the password associated with the specified account on the server. The method establishes a connection to a separate service running on the server, and does not use the POP3 protocol. For this method to succeed, the server must be configured to allow password changes using the "poppass" service, running on port 106.

Because passwords are sent over the network as clear text, this service is considered to be insecure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::CloseMessage Method

```
INT CloseMessage(
   HCLIENT hClient
);
```

The **CloseMessage** method closes the current message that has been opened or created.

Parameters

hClient

Handle to the client session.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If an message is being created, this method actually submits the message to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News messages conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

OpenMessage, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Command Method

```
INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter,
   BOOL bMultiLine
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the class.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

bMultiLine

An optional boolean argument used to specify if multiple lines of data will be returned by the server as the result of the command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker. If this argument is omitted, ***bMultiLine*** is FALSE.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
POP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
POP_OPTION_LINEBREAK	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when retrieving large messages from the server.
POP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
POP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server

	certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
POP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option is the same as specifying POP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STLS command.
POP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the STLS command.
POP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
POP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
POP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
POP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that

thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              POP_PORT_SECURE,
                              POP_TIMEOUT,
                              POP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::DeleteMessage Method

```
INT DeleteMessage(
   UINT nMessage
);
```

The **DeleteMessage** method marks the specified message for deletion from the mailbox.

Parameters

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method only marks the message for deletion. The message is not actually deleted until the user disconnects from the server. To prevent one or more marked messages from actually being deleted from the mailbox, call the **Reset** method to reset the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessage, GetMessageCount, Reset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This

	event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetHeaderValue Method

```
INT GetHeaderValue(
   UINT nMessageId,
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxLength
);

INT GetHeaderValue(
   UINT nMessageId,
   LPCTSTR lpszHeader,
   CString& strValue
);
```

The **GetHeaderValue** method returns the value of a header field in the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the method returns the length of the header field value. If the header field is not present in the message, the method will return a value of zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetHeaderValue** method returns the value of a header field from the specified message. This allows an application to be able to easily determine the value of a header (such as the sender, or the subject of the message) without downloading the entire message header and parsing the contents.

This method uses the XTND XLST command, which is an extension to the POP3 protocol. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this method even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessageHeaders, GetMessageId, GetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessage Method

```
INT GetMessage(
   LONG nMessageId,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

INT GetMessage(
   LONG nMessageId,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

INT GetMessage(
   LONG nMessageId,
   CString& strBuffer
);
```

The **GetMessage** method retrieves the specified message and copies the contents to a local buffer.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpBuffer*** parameter. If the ***lpBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessage** method is used to retrieve an message from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the ***lpBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpBuffer*** parameter point to a global memory handle which will contain the message data when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the complete message has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the POP_EVENT_PROGRESS event

will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the message into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = pClient->GetMessage(nMessageId, &hgblBuffer, &cbBuffer);

if (nResult != POP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // message text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

EnableEvents, GetMessageHeaders, GetTransferStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageCount Method

```
INT GetMessageCount(
   UINT *lpnLastMessage,
   DWORD *lpdwMailboxSize
);
```

The **GetMessageCount** method returns the number of messages available in the mailbox, the last valid message number in the mailbox and the current size of the mailbox in bytes.

Parameters

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. This value will decrease as messages are deleted. If a NULL value is specified, this argument is ignored.

Return Value

If the method succeeds, it returns the number of messages that are currently available. If no messages are available, either because the mailbox is empty or all of the messages have been deleted, this method will return zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

DeleteMessage, GetHeaderValue, GetMessage, GetMessageHeaders, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageHeaders Method

```
INT GetMessageHeaders(
   LONG nMessageId,
   LPBYTE lpHeaders,
   LPDWORD lpdwLength
);

INT GetMessageHeaders(
   LONG nMessageId,
   HGLOBAL *lpHeaders,
   LPDWORD lpdwLength
);

INT GetMessageHeaders(
   LONG nMessageId,
   CString& strHeaders
);
```

The **GetMessageHeaders** method retrieves the headers for the specified message from the server.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpHeaders*** parameter. If the ***lpHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageHeaders** method is used to retrieve an message header block from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpHeaders*** parameter point to a global memory handle which will contain the message headers when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically

fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetHeaderValue, GetMessage, GetMessageId, OpenMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageId Method

```
INT GetMessageId(
   UINT nMessageId,
   LPTSTR lpszMessageId,
   INT nMaxLength
);

INT GetMessageId(
   UINT nMessageId,
   CString& strMessageId
);
```

The **GetMessageId** method returns the message identifier for the specified message.

Parameters

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length. This argument may also be a **CString** object which will contain the message identifier when the method returns.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the length of the unique identifier string. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageId** method returns the message identifier from the Message-ID header of the specified message. The returned value is a typically a string which specifies the domain, date and timestamp for the message that is created when the message is submitted to the mail server for delivery. To obtain a unique identifier for the message in the mailbox, it is recommended that you use the **GetMessageUID** method instead.

This method uses the XTND XLST command to obtain the value of the "Message-ID" header field. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the header value . This enables an application to use this method even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeaderValue, GetMessage, GetMessageHeaders, GetMessageSender, GetMessageUID

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageSender Method

```
INT GetMessageSender(
   UINT nMessageId,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetMessageSender(
   UINT nMessageId,
   CString& strAddress
);
```

The **GetMessageSender** method returns the sender's address for the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender. This argument may also be a **CString** object which will contain the address when the method returns.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the method returns the length of the address. If the sender cannot be determined, the method will return a value of zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageSender** method returns the email address of the user who sent the specified message. This allows an application to be able to easily determine the sender, without downloading the entire header block or contents of the message.

This method uses the XSENDER command, which is an extension to the POP3 protocol, to determine the address of the authenticated sender of the message. If the command is not supported, or the server was unable to authenticate the sender, the method will use the XTND XLST command to obtain the value of the "From" header field. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this method even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeaderValue, GetMessageHeaders, GetMessageId, GetMessageUID

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageSize Method

```
DWORD GetMessageSize(
   UINT nMessageId
);
```

The **GetMessageSize** method returns the size of the specified message.

Parameters

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is the size of the specified message in bytes. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetHeaderValue, GetMessageHeaders, GetMessageId, GetMessageSender, GetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageUID Method

```
INT GetMessageUID(
   UINT nMessageId,
   LPTSTR lpszMessageUID,
   INT nMaxLength
);

INT GetMessageUID(
   UINT nMessageId,
   CString& strMessageUID
);
```

The **GetMessageUID** method returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageUID

Address of a string buffer to receive the unique identifier for the specified message. This should be at least 64 bytes in length. This argument may also be a **CString** object which will contain the UID string when the method returns.

nMaxLength

The maximum length of the string buffer for the message UID.

Return Value

If the method succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the method will return zero. If an error occurs, the method returns POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageUID** method returns the unique message identifier for the specified message. The returned value is a string which can be used to uniquely identify a specific message in the mailbox across multiple client sessions. This is commonly used by mail clients to determine if they have already retrieved a message from the server in a previous session. The UID can also be used as a key or component of the file name to reference the message after it has been stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetHeaderValue, GetMessage, GetMessageHeaders, GetMessageId, GetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetResultCode Method

```
BOOL GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a boolean value, and indicates if the operation succeeded or failed.

Parameters

None.

Return Value

If the previous command was successful, the function returns a non-zero value. If the previous command failed, the function returns zero. To get extended error information, call **GetLastError**.

Remarks

Unlike most other Internet application protocols, the Post Office Protocol does not return numeric result codes to indicate success or failure. If a command is successful, the server will respond with the string "+OK" and this is indicated by the **GetResultCode** method returning a non-zero value. If the command fails, the server will respond with the string "-ERR" along with a description of the error, and this is indicated by the method returning a value of zero. The description of the error returned by the server can be obtained by calling the **GetResultString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

Command, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetResultString Method

```
INT GetResultString(
   LPTSTR lpszResult,
   INT cbResult
);

INT GetResultString(
   CString& strResult
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

The buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
POP_STATUS_UNUSED	No connection has been established.
POP_STATUS_IDLE	The client is current idle and not sending or receiving data.
POP_STATUS_CONNECT	The client is establishing a connection with the server.
POP_STATUS_READ	The client is reading data from the server.
POP_STATUS_WRITE	The client is writing data to the server.
POP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

IsBlocking, IsConnected, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::GetTransferStatus Method

```
INT GetTransferStatus(
   LPPOPTRANSFERSTATUS lpStatus
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an POPTRANSFERSTATUS structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this method will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

EnableEvents, GetStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

CPopClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Login Method

```
INT Login(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nAuthType
);
```

The **Login** method authenticates the specified user in on the server. This method must be called after the connection has been established, and before attempting to retrieve messages or perform any other method on the server.

Parameters

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the current client session. For many service providers, the user name is the full email address of the user which owns the mailbox. In some cases, this may only be the portion of their email address before the domain name.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the current client session. If you are using the POP_AUTH_XOAUTH2 or POP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 bearer token provided by the mail service.

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Value	Description
POP_AUTH_DEFAULT	The default authentication scheme which sends the username and password as cleartext to the server. Because the user credentials are not encrypted, this method should only be used over a secure connection. This is the same as specifying POP_AUTH_PASS as the authentication method.
POP_AUTH_PASS	The username and password is sent to the server using the USER and PASS commands. This authentication method is supported by most servers and is the default authentication type. The credentials are not encrypted and this method should only be used over secure connections.
POP_AUTH_APOP	The APOP authentication method which uses an MD5 digest of the password. This method has been deprecated is not supported by all servers. It should only be used if required by legacy mail servers which do not support the SASL authentication methods.
POP_AUTH_LOGIN	This authentication type will use the LOGIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4422.

POP_AUTH_PLAIN	This authentication type will use the PLAIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
POP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the *lpszPassword* parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
POP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the *lpszPassword* parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The POP_AUTH_LOGIN and POP_AUTH_PLAIN authentication methods require the mail server support the Simple Authentication and Security Layer (SASL) AUTH command as defined in RFC 5034. Most modern mail servers do support one or both of these methods, and they are generally preferred over the POP_AUTH_PASS method when possible. However, for backwards compatibility with legacy servers, the class will default to using POP_AUTH_PASS for client authentication.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The POP_AUTH_XOAUTH2 and POP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetMessage, GetMessageCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::OpenMessage Method

```
INT OpenMessage(
   UINT nMessageId
);
```

The **OpenMessage** method opens the specified message for reading.

Parameters

nMessageId

Number that specifies which message to open. This value must be greater than zero. The first message in the mailbox is message one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **OpenMessage** method is used to begin the process of reading the contents of a message from the from the server. Similar to how a file is opened and read, this method is followed by one or more calls to the **Read** method. When the entire contents of the message has been read, the **CloseMessage** method is used to close the message, completing the transaction on the server.

This is a lower-level method which enables the application to process the message as the contents are being returned by the server. In general, it is recommended that most applications use the **GetMessage** method instead, which provides a simpler interface for retrieving the contents of a message.

It is important to note that you cannot use this method to read the partial contents of a message. Opening a message on the server begins a process where the entire message contents must be read and the message closed before the next command can be issued to the server. If you only want to obtain the headers for a message, use the **GetMessageHeaders** method instead.

Example

```
// Connect to the mail server using the default settings
if (popClient.Connect(strHostName))
{
    // Authenticate the user and display an error if the
    // server does not accept the username or password
    if (popClient.Login(strUserName, strPassword) == POP_ERROR)
    {
        popClient.ShowError();
        popClient.Disconnect();
        return;
    }

    // Open the specified message
    if (popClient.OpenMessage(nMessageId) == POP_ERROR)
    {
        popClient.ShowError();
        popClient.Disconnect();
        return;
    }
```

```
    CString strMessage;
    CString strBuffer;
    INT nResult;

    // Read the contents of the message in a loop; a return value
    // of zero indicates that there is no more data to read
    do
    {
        if ((nResult = popClient.Read(strBuffer)) > 0)
            strMessage = strMessage + strBuffer;
    }
    while (nResult > 0);

    // If an error occurred while reading reading the message,
    // display an error message
    if (nResult == POP_ERROR)
    {
        popClient.ShowError();
        popClient.Disconnect();
        return;
    }

    // Close the message and disconnect from the server
    popClient.CloseMessage();
    popClient.Disconnect();
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

CloseMessage, GetMessage, GetMessageHeaders, IsReadable, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   POPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
POP_EVENT_CONNECT	The connection to the server has completed.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **PopEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **PopEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a message is being retrieved. The POP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one POP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, PopEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Reset Method

```
BOOL Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will prevent any messages marked for deletion from actually being deleted from the mailbox. The client cannot be reset while the client is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

Cancel, DeleteMessage, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::SendMessage Method

```
INT SendMessage(
   LPCTSTR lpszMessage,
   INT nLength
);
```

The **SendMessage** method sends a message to the specified recipients.

Parameters

lpszMessage

Pointer to a string buffer which contains the message to be submitted to the mail server for delivery.

nLength

The length of the string buffer. This specifies the number of characters to be written to the mail server. This value must be greater than zero. If this value is -1, then the length of the string up to the terminating null character is used.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method sends a message through the POP3 server using the XTND XMIT command. The specified file must be in the standard format as described in RFC 822, with the recipient addresses specified in the To: and Cc: header fields. Some servers may support blind carbon copies by using addresses specified in a Bcc: header field, and then removing those addresses from the header before delivering the message.

Note that not all POP3 servers support this command, and it is recommended that you use the Simple Mail Transfer Protocol (SMTP) for general mail delivery purposes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessage, StoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::StoreMessage Method

```
INT StoreMessage(
   UINT nMessageId,
   LPCTSTR lpszFileName
);
```

The **StoreMessage** method stores a message in the specified file.

Parameters

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreMessage** method provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This method always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMessage, GetMessageHeaders, GetTransferStatus, SendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CPopClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol Data Structures

- POPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

POPTRANSFERSTATUS

This structure is used by the GetTransferStatus method to return information about a message transfer in progress.

```
typedef struct _POPTRANSFERSTATUS
{
   UINT    nMessageId;
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} POPTRANSFERSTATUS, *LPPOPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the message is being copied from the server to the local host, this is the size of the message on the server. If the message is being posted to the server, it is the size of message on the local system. If the message size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Remote Command Protocol Class Library

Execute commands on a server or establish an interactive terminal session.

Reference

- Class Methods
- Error Codes

Library Information

Class Name	CRshClient
File Name	CSRSHV12.DLL
Version	12.0.3024.1286
LibID	BA9EA46F-9835-474A-90B1-535762F4E993
Import Library	CSRSHV12.LIB
Dependencies	None
Standards	RFC 1282

Overview

The Remote Command protocol is used to execute a command on a server and return the output of that command to the client. This is most commonly used with UNIX based servers, although there are implementations of remote command servers for the Windows operating system. The class supports both the rcmd and rshell remote execution protocols and provides methods which can be used to search the data stream for specific sequences of characters. This makes it extremely easy to write Windows applications which serve as light-weight client interfaces to commands being executed on a UNIX server or another Windows system. The library can also be used to establish a remote terminal session using the rlogin protocol, which is similar to the Telnet protocol.

The Remote Shell (rsh/rcmd) protocol is deprecated, insecure, and should never be used. This protocol transmits user credentials and command data in plaintext, making it vulnerable to interception, spoofing, and other network-based attacks. Support for this protocol will be removed in a future release. For secure remote command execution and interactive terminal sessions, it is recommended you use the Secure Shell class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum

required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Remote Command Protocol Class Methods

Class	Description
CRshClient	Constructor which initializes the current instance of the class
~CRshClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of network function calls to a file
Execute	Execute a command on the server
FreezeEvents	Suspend asynchronous event processing
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetStatus	Return the current client status
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
Login	Establish a login session with the specified server
Read	Read data returned by the server
RegisterEvent	Register an event callback function
RshEventProc	Callback method that processes events generated by the client
Search	Search for a specific character sequence in the data stream
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error

Write	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::CRshClient

```
CRshClient();
```

The **CRshClient** constructor initializes the class library and validates the license key at runtime.

The Remote Shell (rsh/rcmd) protocol is deprecated, insecure, and should never be used. This protocol transmits user credentials and command data in plaintext, making it vulnerable to interception, spoofing, and other network-based attacks. Support for this protocol will be removed in a future release. For secure remote command execution and interactive terminal sessions, it is recommended you use the Secure Shell class.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **RshInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CRshClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::~CRshClient

```
~CRshClient();
```

The **CRshClient** destructor releases resources allocated by the current instance of the **CRshClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CRshClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CRshClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::AttachHandle

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::AttachThread

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

AttachHandle, Cancel, DetachHandle, Disconnect, Execute, GetHandle, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Cancel

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::DetachHandle

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::DisableEvents

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::DisableTrace

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Disconnect

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

Execute, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::EnableEvents

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This

	event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::EnableTrace

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Execute

```
BOOL Execute(
   LPCTSTR lpszRemoteHost,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommand,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);

BOOL Execute(
   LPCTSTR lpszRemoteHost,
   LPCTSTR lpszUserName,
   LPCTSTR lpszCommand,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Execute** method is used to establish a connection with the server and execute the specified command.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

lpszUserName

A null-terminated string which specifies the username used to authenticate the client session.

lpszPassword

A null-terminated string which specifies the password used to authenticate the client session.

lpszCommand

A null-terminated string which specifies the command to execute on the server.

nRemotePort

The port number the server is listening on. One of the following values should be used:

Value	Description
RSH_PORT_REXEC	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command. This is the default port used when calling the version of the method which specifies a password.
RSH_PORT_RSHELL	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator. This is the default port used when calling the

	version of the method that does not specify a password.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSH_OPTION_RESERVEDPORT	This option specifies that a reserved port should be used to establish the connection. Reserved ports are those port numbers which are less than 1024. This option should be specified when connecting on the RSH_PORT_RSHELL port.
RSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
RSH_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

Return Value

If the method succeeds, the return value non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function

in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Example

The following example demonstrates connecting to a server, executing a command on the server, reading the output and storing it in a string. This uses the rexec service on port 512, and requires a username and password.

```
CRshClient rshClient;
BOOL bConnected;

// Establish a connection with the server and execute
// the command as the specified user
bConnected = rshClient.Execute(strHostName,
                               strUserName,
                               strPassword,
                               strCommand);

// If the connection could not be established or the command
// could not be executed, display an error message
if (bConnected == FALSE)
{
    rshClient.ShowError();
    return;
}

CString strBuffer;
CString strResult;
INT nResult;

// Read the output from the command and store it in the strResult
// string; note that a UNIX server terminates each line of output
// with a bare linefeed, which is something that should be kept
// in mind when parsing the string
do
{
    if ((nResult = rshClient.Read(strBuffer)) > 0)
        strResult += strBuffer;
}
while (nResult > 0);

// If there was an error reading the output from the server, then
// display the error and disconnect
if (nResult == RSH_ERROR)
{
    rshClient.ShowError();
    rshClient.Disconnect();
    return;
}

rshClient.Disconnect();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, Login

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::FreezeEvents

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::GetErrorString

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::GetHandle

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::GetLastError

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::GetStatus

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
RSH_STATUS_UNUSED	No connection has been established.
RSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
RSH_STATUS_CONNECT	The client is establishing a connection with the server.
RSH_STATUS_READ	The client is reading data from the server.
RSH_STATUS_WRITE	The client is writing data to the server.
RSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

IsBlocking, IsConnected, IsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::GetTimeout

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::IsBlocking

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::IsConnected

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::IsInitialized

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

CRshClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::IsReadable

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::IsWritable

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Login

```
BOOL Login(
   LPCTSTR lpszRemoteHost,
   LPCTSTR lpszUserName,
   LPCTSTR lpszTerminal,
   UINT nRemotePort,
   UINT nTimeout
);
```

The **Login** method is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

lpszUserName

A null-terminated string which specifies the username used to authenticate the client session.

lpszTerminal

A null-terminated string which specifies the terminal type which the client will be identified as using during the session. If no particular terminal emulation is required, this parameter may be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port of 513 should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method uses host equivalence, where the client is permitted to login without requiring a password. Host equivalence must be configured by the server administrator and it is typically restricted to specific users. Note that if configured improperly, host equivalence can introduce a significant security loophole. Refer to your UNIX system documentation for more information about host equivalence and the various remote command services.

On UNIX based systems, the terminal name specified by the ***lpszTerminal*** parameter corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this parameter may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this parameter is passed as NULL pointer or an empty string, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available

to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is a using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320. This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, Execute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Read

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::RegisterEvent

```
INT RegisterEvent(
   UINT nEventId,
   RSHEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
RSH_EVENT_CONNECT	The connection to the server has completed.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **RshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **RshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The ***dwParam*** parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, RshEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Search

```
BOOL Search(
   LPCTSTR lpszString
);

BOOL Search(
   LPCTSTR lpszString,
   LPVOID lpBuffer,
   LPDWORD lpdwLength
);

BOOL Search(
   LPCTSTR lpszString,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

BOOL Search(
   LPCTSTR lpszString,
   CString& strBuffer
);
```

The **Search** method searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

lpszString

A null-terminated string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpBuffer

A buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the method returns. An alternate form of this method accepts a **CString** object which will contain the output when the method returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpBuffer*** parameter. If the ***lpBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the ***lpBuffer*** parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds and the character sequences was found in the data stream, the return value is non-zero. If the method fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The method collects the output from the server and stores it in the buffer specified by the ***lpBuffer*** parameter. When the method returns, the buffer will contain

everything sent by the server up to and including the search string.

The ***lpBuffer*** parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the ***lpBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the ***lpBuffer*** parameter point to a global memory handle which will contain the output when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Execute, IsBlocking, IsReadable, Login, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::SetLastError

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::SetTimeout

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

Execute, GetTimeout, IsReadable, IsWritable, Login, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::ShowError

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CRshClient::Write

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Secure Shell Protocol Class Library

Establish an interactive terminal session with an SSH server and execute remote commands.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CSshClient
File Name	CSTSHV12.DLL
Version	12.0.3024.1286
LibID	758D7F52-CE33-4C44-B404-EBF1F15875E8
Import Library	CSTSHV12.LIB
Dependencies	None
Standards	RFC 4251

Overview

The Secure Shell (SSH) protocol is used to establish a secure connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The SshClient class provides an API for establishing the connection and handling the standard I/O functions needed by the program.

Standard password and private key authentication methods are supported. The class will automatically select the strongest cipher suite available to both the client and server, providing support for AES-256 encryption and elliptical curve key exchance algorithms like ECDH and Curve25519. The control also supports some legacy algortithms for backwards compatibility with older servers but will only use them if no other options are available. If the server you are connecting with is based on OpenSSH, we recommend using OpenSSH 7.0 or later. Earlier versions of OpenSSH may request the use of encryption algorithms which are no longer considered secure.

The class also provides methods which enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write lightweight client interfaces to applications running on the server. This class can be combined with the CTerminalEmulator class to provide terminal emulation services for a standard ANSI or DEC-VT220 terminal.

Important: The Secure Shell (SSH) API provides a secure, encrypted alternative to legacy protocols such as Remote Shell (rsh/rcmd), Remote Login (rlogin), and Telnet. Unlike those older protocols, SSH uses strong authentication and end-to-end encryption to protect user credentials, command input, and output data from interception or tampering. SSH is the recommended protocol for remote command execution and interactive terminal sessions, particularly when connecting over public or untrusted networks. It ensures confidentiality and integrity of data in transit, and is widely supported on modern operating systems and network infrastructure. If you are currently using Telnet or Remote Shell in your applications, you are strongly encouraged to migrate to the Secure Shell API. These older

protocols are inherently insecure and should not be used in any context where data security matters.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SSH Protocol Class Methods

Class	Description
CSshClient	Constructor which initializes the current instance of the class
~CSshClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Break	Send a break signal to the server
Cancel	Cancel the current blocking operation
Connect	Connect to the specified server
Control	Send a control message to the server
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of network function calls to a file
Execute	Execute a command on a server and return the output in the specified buffer
FreezeEvents	Suspend asynchronous event processing
GetErrorString	Return a description for the specified error code
GetExitCode	Return the exit code for the remote session
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetLineMode	Return the current mode used to send end-of-line character sequences
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server

Peek	Examine data in the receive buffer, but do not remove it
Read	Read data returned by the server
ReadLine	Read a line of data from the server and return it in a string buffer
RegisterEvent	Register an event callback function
Search	Search for a specific character sequence in the data stream
SetLastError	Set the last error code
SetLineMode	Change how end-of-line character sequences are sent to the server
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
SshEventProc	Callback function that processes events generated by the client
Write	Write data to the server
WriteLine	Write a line of data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::CSshClient Method

```
CSshClient();
```

The **CSshClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **SshInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CSshClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::~CSshClient

```
~CSshClient();
```

The **CSshClient** destructor releases resources allocated by the current instance of the **CSshClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CSshClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CSshClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Break Method

```
INT Break();
```

The **Break** method sends a signal to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Break** method sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, Control, Read, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

Break, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nTimeout,
   DWORD dwOptions,
   LPSSHOPTIONDATA lpOptions
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A null-terminated string which specifies the user name which will be used to authenticate the client session. This parameter must specify a valid user name and cannot be NULL or an empty string.

lpszPassword

A null-terminated string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this parameter can be NULL or an empty string.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPALIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for an extended period of time. The actual client idle timeout period is determined by the server configuration.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified. This option should never be used with interactive client sessions.

SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server. You should not use this option unless you require the command to execute without the normal shell environment. In most cases you should not use this option because it may cause commands to fail unexpectedly.
SSH_OPTION_TERMINAL	This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.
SSH_OPTION_COMMAND	This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudoterminal is created for the client. The szCommandLine member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.
SSH_OPTION_PROXYSERVER	This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-related members of the SSHOPTIONDATA structure should be set to the appropriate values.
SSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpOptions

A pointer to a SSHOPTIONDATA structure which specifies additional information for one or more options. If no optional data is required, a NULL pointer may be specified.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session

will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

By default, the client session will be authenticated using standard password authentication. If you prefer to authenticate using a private key file, you must create a SECURITYCREDENTIALS structure which specifies the path to the private key and optionally the password used when creating it. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported. To determine how the private key was created, you can open the file in a text editor and the first line of the file will identify the format.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the SSH_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Example

```
CSshClient sshClient;
SSHOPTIONDATA sshOptions;
BOOL bConnected;

// Initialize the SSHOPTIONDATA structure and specify the
// command that should be executed on the server
ZeroMemory(&sshOptions, sizeof(sshOptions));
lstrcpyn(sshOptions.szCommandLine, strCommand, SSH_MAXCOMMANDLEN);

// Establish a connection with the SSH server

bConnected = sshClient.Connect(strHostName,
                               SSH_PORT_DEFAULT,
                               strUserName,
```

```
                                    strPassword,
                                    SSH_TIMEOUT,
                                    SSH_OPTION_COMMAND,
                                    &sshOptions);

// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (!bConnected)
{
    sshClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials, Disconnect, GetSecurityInformation, SECURITYCREDENTIALS, SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshClient::Control Method

```
INT Control(
   DWORD dwControlCode
);
```

The **SshControl** function sends a control message to the server.

Parameters

dwControlCode

A numeric control code which specifies the control message which should be sent to the server. This may be one of the following values:

Value	Description
SSH_CONTROL_BREAK	Sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.
SSH_CONTROL_NOOP	Sends a control message to the server, but it does not perform any operation. This is typically used by clients to prevent the server from automatically closing a session that has been idle for a long period of time.
SSH_CONTROL_EOF	Sends a control message to the server indicating that the client has finished sending data. Note that this option is normally not used with interactive terminal sessions, and should only be used when required by the server.
SSH_CONTROL_PING	Sends a control message to the server which is used to test whether or not the server is responsive to the client. This is typically used by clients to attempt to detect if the connection to the server is still active.
SSH_CONTROL_REKEY	Sends a control message to the server requesting that the key exchange be performed again. This control code is ignored for SSH 1.0 sessions.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **Control** method enables an application to send control messages to the server, which can cause it to take specific actions such as simulate a terminal break or request that the key exchanged be performed again. Some control messages are not supported by the SSH 1.0 protocol, in which case the control message is ignored.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

Break, Cancel, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszKeyFile,
   LPCTSTR lpszPassword
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszKeyFile,
   LPCTSTR lpszPassword
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszKeyFile
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols.

Value	Description
SECURITY_PROTOCOL_SSH2	Version 2.0 of the protocol will be used when establishing the connection. Currently, this is the only supported protocol option for SSH connections. In older versions of SocketTools, you had the option of selecting either version 1.0 or 2.0; however, version 1.0 is no longer considered secure and is not supported.

dwOptions

This parameter is reserved for future use and should always be a value of zero.

lpszKeyFile

A null-terminated string which specifies the name of a private key file that used when authenticating the client connection. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported.

lpszPassword

A null-terminated string which specifies the password for the private key file. A value of NULL specifies that no password is required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is only required if you wish to authenticate the client session using a private key file instead of an account password.

The ***lpszKeyFile*** path can use environment variables enclosed in percent symbols, and the path to the

private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results with multithreaded applications because the current working directory for a process is a global value and it can be changed by any thread at any time.

Example

```
CSshClient sshClient;
LPCTSTR lpszKeyFile = _T("%USERPROFILE%\\.ssh\\id_rsa.pem");

// Create the SECURITYCREDENTIALS structure and specify the
// path to the private key file for the current user
if (!sshClient.CreateSecurityCredentials(lpszKeyFile))
{
    sshClient.ShowError();
    return;
}

// Establish a connection with the SSH server on port 22
bConnected = sshClient.Connect(strHostName,
                               SSH_PORT_DEFAULT,
                               strUserName,
                               NULL, // No password, using a private key file
                               SSH_TIMEOUT,
                               SSH_OPTION_KEEPALIVE);

// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (!bConnected)
{
    sshClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.

SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Execute Method

```
INT Execute(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommandLine,
   UINT nTimeout,
   DWORD dwOptions,
   HGLOBAL* lphgblBuffer,
   LPDWORD lpdwLength
);

INT Execute(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommandLine,
   UINT nTimeout,
   DWORD dwOptions,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

INT Execute(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommandLine,
   UINT nTimeout,
   DWORD dwOptions,
   CString& strBuffer
);
```

The **Execute** method executes a command on the server and returns the output in the specified buffer.

Parameters

lpszRemoteHost

A null-terminated string which specifies the name of the server. This may either be a fully-qualified domain name, or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A null-terminated string which specifies the user name which will be used to authenticate the client session.

lpszPassword

A null-terminated string which specifies the password which will be used to authenticate the client

session.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPALIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.

lpBuffer, lphgblBuffer or strBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns. If the application is using MFC, then a CString variable may also be specified, in which case the data is returned in the string

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function.

Return Value

If the function succeeds, the return value is the exit code from the program that was executed on the server. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **Execute** method is used to execute a command on a server, read the output from that command and copy it into a local buffer. This method cannot be used if the connection to the server must be established through a proxy server; if a proxy server must be used, then you should use the **Connect** method to establish the connection, and then use either the **Read** or **ReadLine** methods to read the output.

This method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the command output. In this case, the ***lpBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lphgblBuffer*** parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

When the command output is being read from the server, this method will automatically convert the data to match the end-of-line convention used on the Windows platform. This is useful when executing a command on a UNIX based system where the end-of-line is indicated by a single linefeed, while on Windows it is a carriage-return and linefeed pair. If the output contains embedded nulls or escape sequences, then this conversion will not be performed.

This method will cause the current thread to block until the command completes or a timeout occurs.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Execute a command on the server and return the data into block
// of global memory allocated by the GlobalAlloc function; the handle
```

```
// to this memory will be returned in the hgblBuffer parameter
nResult = sshClient.Execute(strHostName,
                            SSH_PORT_DEFAULT,
                            strUserName,
                            strPassword,
                            strCommandLine,
                            SSH_TIMEOUT,
                            SSH_OPTION_NONE,
                            &hgblBuffer,
                            &cbBuffer);

if (nResult != SSH_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)::GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    ::GlobalUnlock(hgblBuffer);
    ::GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Server: Requires Windows Server 2003 or Windows 2000 Server.
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetExitCode, Read, ReadLine, Write, WriteLine, SECURITYCREDENTIALS, SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetExitCode Method

```
INT GetExitCode();
```

The **GetExitCode** function returns the exit code for the remote session.

Parameters

None.

Return Value

If the method succeeds, the return value is the numeric exit code. If the function fails, the return value is SSH_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

This method should only be called after the command has completed and the **Read** method has returned a value of zero. In most cases, an exit code value of zero indicates success, while any other value indicates an error condition.

Note that the actual value is application dependent and is only meaningful in the context of that particular program. A program may choose or use exit codes in a non-standard way, such as having certain non-zero values indicate success.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetLineMode Method

```
INT GetLineMode();
```

The **GetLineMode** method returns the current line mode.

Parameters

None.

Return Value

If the method succeeds, the return value is the current line mode. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetLineMode** method returns an integer value that specifies how end-of-line character sequences are sent to the server. For more information about how newlines are processed by the class and the available options, refer to the **SetLineMode** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SetLineMode, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Get the server fingerprint
SECURITYINFO securityInfo;

ZeroMemory(&securityInfo, sizeof(SECURITYINFO));
securityInfo.dwSize = sizeof(SECURITYINFO);

if (sshClient.GetSecurityInformation(&securityInfo))
{
    if (securityInfo.lpszFingerprint != NULL)
    {
        TCHAR szMessage[256];
        wsprintf(szMessage, _T("The fingerprint is %s",
securityInfo.lpszFingerprint);
        MessageBox(NULL, szMessage, "Connection", MB_OK);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
SSH_STATUS_UNUSED	No connection has been established.
SSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
SSH_STATUS_CONNECT	The client is establishing a connection with the server.
SSH_STATUS_AUTHENTICATE	The client is authenticating the session with the server.
SSH_STATUS_READ	The client is reading data from the server.
SSH_STATUS_WRITE	The client is writing data to the server.
6	SSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

IsBlocking, IsConnected, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

CSshClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 8192 bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Peek Method

```
INT Peek(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Peek(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Peek** method reads the specified number of bytes from the server and copies them into the buffer, but it does not remove the data from the internal receive buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method can be used to examine the data that is available to be read from the internal receive buffer. If there is no data in the receive buffer at that time, a value of zero is returned. It should be noted that this differs from the **Read** method, where a return value of zero indicates that there is no more data available to be read and the connection has been closed. The **Peek** method will never cause the client to block, and so may be safely used with asynchronous connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

IsBlocking, IsReadable, Read, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

IsBlocking, IsReadable, Peek, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::ReadLine Method

```
BOOL ReadLine(
   LPTSTR lpszBuffer,
   LPINT lpnLength
);

BOOL ReadLine(
   LPTSTR lpszBuffer,
   INT nMaxLength
);

BOOL ReadLine(
   CString& strBuffer,
   INT nMaxLength
);
```

The **ReadLine** method reads up to a line of data from the server and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the server and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the socket connection. If this method is called

with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pClient->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pClient->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsReadable, Peek, Read, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   SSHEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
SSH_EVENT_CONNECT	The connection to the server has completed.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **SshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The ***dwParam*** parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, SshEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Search Method

```
BOOL Search(
   LPCTSTR lpszString
);

BOOL Search(
   LPCTSTR lpszString,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

BOOL Search(
   LPCTSTR lpszString,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

BOOL Search(
   LPCTSTR lpszString,
   CString& strBuffer
);
```

The **Search** method searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

lpszString

A null-terminated string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpBuffer

A buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the method returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpBuffer*** parameter. If the ***lpBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the ***lpBuffer*** parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds and the character sequences was found in the data stream, the return value is non-zero. If the method fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as logging in a user and executing a command. The method collects the output from the server and stores it in the buffer specified by the ***lpBuffer*** parameter. When the method returns, the buffer will contain everything sent by the server up to and including the search string.

The *lpBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the output when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
LPCTSTR lpszCommand = _T("/bin/ls -l\r\n");
LPCTSTR lpszPrompt = _T("$ ");
HGLOBAL hgblOutput = NULL;
DWORD cbOutput = 0;
BOOL bResult;

// Search for a command prompt issued by the server

bResult = pClient->Search(lpszPrompt, NULL, NULL, 0);

// If the shell prompt was found, issue the command
// and capture the output into the hgblBuffer global
// memory buffer; the cbBuffer variable will contain
// the actual number of bytes in the buffer when the
// function returns

if (bResult)
{
    pClient->Write((LPBYTE)lpszCommand, lstrlen(lpszCommand));

    bResult = pClient->Search(lpszPrompt,
                              &hgblOutput,
                              &cbOutput,
                              0);
}

// Write the contents of the output buffer to the
// standard output stream

if (bResult)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    if (lpBuffer)
        fwrite(lpBuffer, 1, cbBuffer, stdout);

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsBlocking, IsReadable, Peek, Read, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::SetLineMode Method

```
INT SetLineMode(
   INT nLineMode
);
```

The **SetLineMode** method changes the current line mode for the client session.

Parameters

nLineMode

An integer value which specifies how the newlines are sent by the class. It must be one of the following values:

Value	Description
SSH_NEWLINE_DEFAULT	There are no changes to how data is sent to the server. Any carriage return or linefeed characters that are sent using the **Write** method will be sent as-is. The **WriteLine** method will terminate each line of text with a carriage return and linefeed (CRLF) sequence. This is the default line mode that is set when a new connection is established.
SSH_NEWLINE_CR	A carriage return is used as the end-of-line character. Any data sent using the **Write** method that contains only a linefeed (LF) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a carriage return (CR) character. The **WriteLine** method will terminate each line of text with a single carriage return character.
SSH_NEWLINE_LF	A linefeed is used as the end-of-line character. Any data sent using the **Write** method that contains only a carriage return (CR) character or a carriage return an linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a linefeed (LF) character. The **WriteLine** method will terminate each line of text with a single linefeed character.
SSH_NEWLINE_CRLF	A carriage return and linefeed (CRLF) character sequence is used to indicate the end-of-line. Any data sent using the **Write** method that contains only a carriage return (CR) or linefeed (LF) will be replaced by a carriage return and linefeed. The **WriteLine** method will terminate each line of text with a carriage return and linefeed sequence.

Return Value

If the method succeeds, the return value is the previous line mode for the client session. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a connection is initially established with the server, it determines what characters are used to indicate the end-of-line and how they are displayed. On UNIX based systems, this is controlled by the settings for the pseudo-terminal that is allocated for the client session, and can be changed using the **stty** command. In most cases, the client line mode can be left at the default. However, in some cases

you may need to change the line mode, particularly if you intend to send data from a Windows text file or copied from the clipboard.

Windows uses a carriage return and linefeed (CRLF) sequence to indicate the end-of-line and a UNIX based server may interpret that as multiple newlines. To prevent this, use the **SetLineMode** method to change the current line mode to **SSH_NEWLINE_CR** and the CRLF sequence in the text will be replaced by a single carriage return.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

GetLineMode, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsBlocking, IsWritable, Peek, Read, ReadLine, SetLineMode, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSshClient::WriteLine Method

```
BOOL WriteLine(
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **WriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each

connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsWritable, Peek, Read, ReadLine, SetLineMode, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SSH Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO
- SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure.

dwProtocol

A value which specifies which security protocols are supported:

Value	Description
SECURITY_PROTOCOL_SSH2	Version 2.0 of the protocol will be used when establishing the connection. Currently, this is the only supported protocol option for SSH connections. In older versions of SocketTools, you had the option of selecting either version 1.0 or 2.0; however, version 1.0 is no longer considered secure and is not supported.

dwOptions

This structure member is reserved for use with TLS connections and should always be initialized to zero.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating a server certificate. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this

member is only used if a private key file has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. Although this member can be used to specify the path to a public key file in the OpenSSH format, this is not required. In most cases this member should be initialized as a NULL pointer for connections using the SSH protocol.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszKeyFile

A null-terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used with the SSH protocol. If the member is NULL, then no private key is used. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported.

Remarks

When using the C++ class, you do not need to explicitly define this structure. Instead, you can use the **CreateSecurityCredentials** method to specify the private key file. If the private key is protected with a password, the ***lpszPassword*** parameter must specify that value or the connection request will fail with an authentication error.

The ***lpszKeyFile*** path can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results with multithreaded applications because the current working directory for a process is a global value and it can be changed by any thread at any time.

Example

```
CSshClient sshClient;
LPCTSTR lpszKeyFile = _T("%USERPROFILE%\\.ssh\\id_rsa.pem");

// Create the SECURITYCREDENTIALS structure and specify the
// path to the private key file for the current user
if (!sshClient.CreateSecurityCredentials(lpszKeyFile))
{
    sshClient.ShowError();
    return;
}

// Establish a connection with the SSH server on port 22
bConnected = sshClient.Connect(strHostName,
                               SSH_PORT_DEFAULT,
                               strUserName,
                               NULL, // No password, using a private key file
                               SSH_TIMEOUT,
                               SSH_OPTION_KEEPALIVE);

// If the connection attempt fails, then get a description of
// the error and display it in a message box
```

```
if (!bConnected)
{
    sshClient.ShowError();
    return;
}
```

In this example, the path to the private key file will be expanded to an absolute path because the USERPROFILE environment variable defines the home directory for the current user. The path must resolve to a normal file which that can be read by the current process. If the file does not exist, or it does not specify a PEM formatted file which contains an RSA or OpenSSH private key, the connection will fail and the last error code will be set to `ST_ERROR_INVALID_PRIVATE_KEY`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, DeleteSecurityCredentials, GetSecurityInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and fingerprint.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol was selected.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Value	Description
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.

SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

This member is not used with SSH connections and will have a value of zero.

stCertIssued

This member is not used with SSH connections and will have a value of zero.

stCertExpires

This member is not used with SSH connections and will have a value of zero.

lpszCertIssuer

This member is not used with SSH connections and will have a value of zero.

lpszCertSubject

This member is not used with SSH connections and will have a value of zero.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and cannot be used when establishing a connection with the server. Although SSH protocol version 1.0 is defined in the header file, it is important to note that it is considered insecure and is no longer supported.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SSHOPTIONDATA

The **SSHOPTIONDATA** structure specifies additional option information for the client session.

```
typedef struct _SSHOPTIONDATA
{
    DWORD dwSize;
    DWORD dwReserved;
    UINT  nProxyType;
    UINT  nProxyPort;
    TCHAR szProxyHost[128];
    TCHAR szProxyUser[128];
    TCHAR szProxyPassword[128];
    UINT  nTermCols;
    UINT  nTermRows;
    TCHAR szTermName[32];
    TCHAR szCommandLine[512];
} SSHOPTIONDATA, *LPSSHOPTIONDATA;
```

Members

dwSize

An unsigned integer value which specifies the size of the SSHOPTIONDATA structure. This member must be initialized prior to passing the structure to the **Connect** method.

dwReserved

An unsigned integer value that is reserved for internal use, and should always be initialized to a value of zero.

nProxyType

An unsigned integer value that specifies the type of proxy that the client should connect through. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified. Possible values are:

Value	Description
SSH_PROXY_NONE	No proxy server should be used when establishing the connection.
SSH_PROXY_HTTP	The connection should be established on port 80 using HTTP. An alternate port number can be specified by setting the *nProxyPort* structure member to the desired value.
SSH_PROXY_TELNET	The connection should be established on port 23 using TELNET. An alternate port number can be specified by setting the *nProxyPort* structure member to the desired value.

nProxyPort

An unsigned integer value that specifies the port number which should be used to establish the proxy connection. A value of zero specifies that the default port number appropriate for the selected protocol should be used. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyHost

A null-terminated string which specifies the host name or IP address of the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyUser

A null-terminated string which specifies the user name which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyPassword

A null-terminated string which specifies the password which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

nTermCols

An unsigned integer value which specifies the number of columns for the virtual terminal allocated for the client session. The default number of columns is 80. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

nTermRows

An unsigned integer value which specifies the number of rows for the virtual terminal allocated for the client session. The default number of rows is 25. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szTermName

A null-terminated string which specifies the name of the terminal emulation type. On UNIX based systems, this name typically corresponds to an entry in the terminal capability database (either termcap or terminfo). If the name is not specified, then the default name terminal name of "unknown" will be used. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szCommandLine

A null-terminated string which specifies the command that should be executed on the server. The output from the command is returned to the client, and the session is terminated. This structure member is only used if the option SSH_OPTION_COMMAND has been specified.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol Class Library

Submit email messages for delivery to one or more recipients.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CSmtpClient
File Name	CSMTPV12.DLL
Version	12.0.3024.1286
LibID	7EFD241B-CEB0-4667-8944-4231A1B46064
Import Library	CSMTPV12.LIB
Dependencies	None
Standards	RFC 821, RFC 1425, RFC 1869, RFC 2821

Overview

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The class library provides an interface for addressing and delivering messages, and extended features such as user authentication and delivery status notification. Unlike Microsoft's Messaging API (MAPI) or Collaboration Data Objects (CDO), there is no requirement to have certain third-party email applications installed or specific types of servers installed on the local system. The CSmtpClient class can be used to deliver mail through a wide variety of systems, from standard UNIX based mail servers to Windows systems running Microsoft Exchange.

Using this class library, messages can be delivered directly to the recipient, or they can be routed through a relay server, such as an Internet service provider's mail system. The CMailMessage class can be combined with this class to provide a flexible interface for composing and delivering messages.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of

Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol Class Methods

Class	Description
CSmtpClient	Constructor which initializes the current instance of the class
~CSmtpClient	Destructor which releases resources allocated by the class
Method	Description
AddRecipient	Add an address to the recipient list
AppendMessage	Append contents of specified file to the current message
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Authenticate	Authenticate the client session with a user name and password
Cancel	Cancel the current blocking operation
CloseMessage	Close the message being composed and submit for delivery
Command	Send a command to the server
Connect	Establish a connection with a server
CreateMessage	Create a new message
CreateSecurityCredentials	Allocate a structure to establish client security credentials
DeleteSecurityCredentials	Delete the specified client security credentials
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable all event notification, including event callbacks
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable event handling by the library
EnableTrace	Enable logging of network function calls to a file
ExpandAddress	Expand the specified address
FreezeEvents	Suspend and resume event handling by the client
GetCurrentDate	Return the current date and time
GetDeliveryOptions	Return the delivery options for the current session
GetErrorString	Return a description for the specified error code
GetExtendedOptions	Return the extended options supported by the server
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetLocalName	Return the local host name assigned for the client session
GetResultCode	Return the result code from the previous command

GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current status of the client
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
RegisterEvent	Register an event handler for the specified event
Reset	Reset the client and return to a command state
SendMessage	Send message to the specified recipient
SetDeliveryOptions	Set the delivery options for the current session
SetLastError	Set the last error code
SetLocalName	Set the local host name to be used for the client session
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
SmtpEventProc	Process events generated by the client
SubmitMessage	Compose and submit a message for delivery to the specified mail server
VerifyAddress	Verify that the specified address is valid
Write	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::CSmtpClient Method

```
CSmtpClient();
```

The **CSmtpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **SmtpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CSmtpClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::~CSmtpClient

```
~CSmtpClient();
```

The **CSmtpClient** destructor releases resources allocated by the current instance of the **CSmtpClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CSmtpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CSmtpClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::AddRecipient Method

```
INT AddRecipient(
   LPCTSTR lpszAddress
);
```

The **AddRecipient** method adds the specified address to the recipient list for the current message. This method should be called once for each recipient.

Parameters

lpszAddress

Points to a string which specifies the address to be added to the recipient list.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseMessage, CreateMessage, ExpandAddress, VerifyAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::AppendMessage Method

```
INT AppendMessage(
   LPCTSTR lpszFileName
);

INT AppendMessage(
   LPBYTE lpMessage,
   DWORD dwMessageSize
);

INT AppendMessage(
   HGLOBAL hgblMessage,
   DWORD dwMessageSize
);
```

The **AppendMessage** method writes the contents of a specified file or buffer to the data stream, appending it to the current message contents.

Parameters

lpszFileName

A null-terminated string which specifies the name of a file. The contents of the file are appended to the current message being submitted to the mail server for delivery.

lpMessage

Pointer to a buffer which contains the message data to be appended to the current message.

hgblMessage

A global memory handle which references data that is to be appended to the current message.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **AppendMessage** method is used to append the contents of a file, memory buffer, or global memory handle to the current message that is being composed for delivery. To send a complete RFC 822 formatted message, refer to the **SendMessage** method.

This method will cause the current thread to block until the complete message has been written, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseMessage, CreateMessage, SendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Authenticate Method

```
INT Authenticate(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nAuthType
);
```

The **Authenticate** method provides client authentication information to the server.

Parameters

lpszUserName

A null-terminated string which specifies the account name for the user authorized to send mail through the server.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the current client session. If you are using the SMTP_AUTH_XOAUTH2 or SMTP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the bearer token provided by the mail service.

nAuthType

An integer value which specifies which method the library should use to authenticate the client session. This parameter should be set to one of the following values:

Value	Description
SMTP_AUTH_LOGIN	The client will authenticate using the AUTH LOGIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. This is the default method accepted by most mail servers and is the preferred authentication type for most clients.
SMTP_AUTH_PLAIN	The client will authenticate using the AUTH PLAIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
SMTP_AUTH_XOAUTH2	The client will authenticate using the AUTH XOAUTH2 command. This authentication method does not require the user password, instead the ***lpszPassword*** parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
SMTP_AUTH_BEARER	The client will authenticate using the AUTH OAUTHBEARER command as defined in RFC 7628. This authentication method does not require the user password, instead the ***lpszPassword*** parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.

Return Value

If the method succeeds, the return value is the command result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

To submit a mail message for delivery, virtually all public mail servers require clients to authenticate and will only accept messages from authorized users. In some cases, they may also require the sender email address match the account being used to authenticate the session. It is also typical for most public mail servers to reject authentication attempts over a standard (non-secure) connection. You should always use a secure connection whenever possible.

All authentication methods require the mail server to support the standard service extensions for authentication as specified in the RFC 4954. The server must support the ESMTP protocol extensions and the AUTH command. A user name and password are required for authentication. If you wish to authenticate without a user password, you must use one of the OAuth 2.0 authentication methods.

If the *nAuthType* parameter is omitted, it will default to using SMTP_AUTH_LOGIN and this is accepted by most mail servers. It is common for mail servers to allow the SMTP_AUTH_PLAIN method as well, however it is recommended you explicitly check whether the server supports the desired authentication method by calling the **GetExtendedOptions** method. If you attempt to use an authentication type which is not supported by the server, this method will fail and the last error code will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The SMTP_AUTH_XOAUTH2 and SMTP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Example

```
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server

BOOL bExtended = pClient->GetExtendedOptions(&dwOptions);

if (bUseBearerToken)
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
```

```
    {
        INT nResult = pClient->Authenticate(lpszUserName, lpszBearerToken,
SMTP_AUTH_XOAUTH2);

        if (nResult == SMTP_ERROR)
        {
            // An error occurred during authentication; when using an
            // OAuth 2.0 bearer token, this typically means that the token
            // has expired and must be refreshed
            return;
        }
    }
    else
    {
        // The server does not support XOAUTH2
        return;
    }
}
else
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_AUTHLOGIN))
    {
        INT nResult = pClient->Authenticate(lpszUserName, lpszPassword);

        if (nResult == SMTP_ERROR)
        {
             // An error occurred during authentication
            return;
        }
    }
    else
    {
         // The server does not support AUTH LOGIN
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetExtendedOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::CloseMessage Method

```
INT CloseMessage();
```

The **CloseMessage** method ends the composition of the current message. The server then queues the message for delivery to each recipient specified by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CloseMessage** method should be called after all of the message data has been written to the data stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AppendMessage, CreateMessage, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Command Method

```
INT Command(
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the API.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the command was successful, the method returns the result code. If the command failed, the method returns SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetResultCode, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit TLS connections, the default port number is 465.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the SMTP_OPTION_EXTENDED option must be specified.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number,

	default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
SMTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
SMTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SMTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when

a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the SMTP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Authenticate, Disconnect, GetExtendedOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::CreateMessage Method

```
INT CreateMessage(
   LPCTSTR lpszSender,
   DWORD dwMessageSize
);
```

The **CreateMessage** method creates a new message for delivery.

Parameters

lpszSender

A null-terminated string which specifies the email address of the user sending the message. This typically corresponds to the address in the From header of the message, but it is not required that they be the same.

dwMessageSize

An unsigned integer which specifies the size of the message in bytes. If the size of the message is unknown, this value should be zero. This parameter is ignored if the server does not support extended features. If the message size is larger than what the server will accept, this method will fail. Most Internet Service Providers impose a limit on the size of an email message, typically between 5 and 10 megabytes.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CreateMessage** method begins the composition of a new message to be submitted to the mail server for delivery. There are several steps that must be followed when dynamically composing a message for delivery:

1. Call the **CreateMessage** method to begin the message composition. The sender email address should generally be the same address as the one used in the "From" header field in the message.
2. Call the **AddRecipient** method for each recipient of the message. These addresses are typically specified in the "To" and "Cc" header fields in the message. Additional addresses may also be be provided which are not specified in the email message itself. This is how one or more blind carbon copies of a message is delivered. Most servers have a limit on the total number of recipients that may be specified for a single message. This limit is usually around 100 addresses.
3. Call the **Write** method to write the contents of the message to the data stream. The application may also choose to use the **AppendMessage** method to write out a large amount of message data, or write the contents of a file to the data stream.
4. Call the **CloseMessage** method to close the message and submit it to the mail server for delivery.

For applications that do not need to dynamically compose the message and already have the message contents stored in a file or memory buffer, the **SendMessage** method is the preferred method of submitting a message for delivery.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddRecipient, AppendMessage, CloseMessage, SendMessage, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              SMTP_PORT_SECURE,
                              SMTP_TIMEOUT,
                              SMTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This

	event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SMTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
SMTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
SMTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::ExpandAddress Method

```
INT ExpandAddress(
   LPCTSTR lpszMailingList,
   LPTSTR lpszAddresses,
   INT nMaxLength
);

INT ExpandAddress(
   LPCTSTR lpszMailingList,
   CString& strAddresses
);
```

The **ExpandAddress** method expands the specified mailing list, returning the membership of that list.

Parameters

lpszMailingList

Points to a string which specifies the mailing list that the server should expand into full addresses.

lpszAddresses

Points to a buffer that the expanded addresses will be copied into. This argument may also be a **CString** object which will contain the expanded addresses when the method returns.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ExpandAddress** method requests that the server expand the specified email address. Typically this is used to expand aliases which refer to a mailing list, returning all of the members of that list. A server may not support this command, or may restrict its usage. An application should not depend on the ability to expand addresses.

This method cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddRecipient, VerifyAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetCurrentDate Method

```
INT GetCurrentDate(
   LPTSTR lpszDateString,
   INT nMaxLength
);

INT GetCurrentDate(
   CString& strDateString
);
```

The **GetCurrentDate** method copies the current date and time to the specified buffer in a format that is commonly used in mail messages. This date format should be used in all date-related fields in the message header.

Parameters

lpszDateString

Pointer to a string buffer that will contain the current date and time when the method returns. This argument may also be a **CString** object.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseMessage, CreateMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetDeliveryOptions Method

```
BOOL GetDeliveryOptions(
   LPDWORD lpdwOptions
);
```

The **GetDeliveryOptions** method returns the delivery status notification options for the current session.

Parameters

lpdwOptions

Address of a variable that will be set to the current delivery options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Value	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDeliveryOptions** method returns the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

Connect, GetExtendedOptions, SetDeliveryOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetExtendedOptions Method

```
BOOL GetExtendedOptions(
   LPDWORD lpdwOptions
);
```

The **GetExtendedOptions** method returns the extended server options for the current session.

Parameters

lpdwOptions

Address of a variable that will be set to the current server options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Value	Description
SMTP_EXTOPT_EXPN	The server supports address expansion using the EXPN command. The **ExpandAddress** method can be used to expand addresses, which typically returns the email addresses associated with a mailing list. Most public mail servers restrict or disable this functionality because it can present a security risk. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_VRFY	The server supports verification of addresses using the VRFY command. The **VerifyAddress** method can be used to verify addresses. Most public mail servers restrict the ability for clients to verify email addresses to prevent potential abuse. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_DSN	The server supports delivery status notification (DSN) which allows the sender to be notified when a message has been delivered, or when an error occurs during the delivery process. The **SetDeliveryOptions** method can be used to specify the delivery options to be used in the current session.
SMTP_EXTOPT_SIZE	The server supports the use of the SIZE parameter, which enables the client to determine the maximum message size that may be delivered through the server. Most public mail servers impose a limit of on the total size of a message, including any encoded attachments.
SMTP_EXTOPT_ETRN	The server supports the use of the ETRN command, instructing the server to start processing its message queues for a specific host. Most public mail servers do not support this capability and its use has been deprecated.
SMTP_EXTOPT_8BITMIME	The server supports the delivery of messages that contain characters with the high bit set. Most servers support this option, however it is recommended that you encode any message text which contains non-ASCII characters to ensure the broadest compatibility with other servers and clients.
SMTP_EXTOPT_STARTTLS	The server supports explicit TLS sessions. This extended option

	is used internally to determine how secure connections should be established, and if a secure connection can be made using the standard submission port.
SMTP_EXTOPT_UTF8	The server supports UTF-8 encoding in email addresses and the message envelope. Not all mail servers will have this extended capability enabled, and applications should not depend on being able to provide internationalized user and domain names unless this option bit flag has been set.

In addition, there are extended options which specify the authentication methods supported by the server. A server will typically support multiple authentication methods and may be one or more of the following values:

Value	Description
SMTP_EXTOPT_AUTHLOGIN	The server supports client authentication using the AUTH LOGIN command. This is the default authentication method and is supported by most mail servers. The user name and password are encoded in a specific format, but are not encrypted. The client should use a secure connection whenever possible.
SMTP_EXTOPT_AUTHPLAIN	The server supports client authentication using the AUTH PLAIN command. The use name and password are encoded in a specific format, but are not encrypted. If a server supports this authentication method, it is very likely it also supports AUTH LOGIN. It is recommended you use only use AUTH PLAIN authentication if the server does not support AUTH LOGIN.
SMTP_EXTOPT_XOAUTH2	The server supports client authentication using AUTH XOAUTH2 command. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method.
SMTP_EXTOPT_BEARER	The server supports client authentication using AUTH OAUTHBEARER command as specified in RFC 7628. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method. The connection must be secure to use this authentication method.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetExtendedOptions** method returns the extended options supported by the server. The use of

extended options requires that the server support the ESMTP protocol, and that the client connect using the SMTP_OPTION_EXTENDED option.

You should check these options prior to calling **Authenticate** to determine which authentication methods are acceptable to the server. If you wish to use an OAuth 2.0 bearer token, always check to make sure either the SMTP_EXTOPT_XOAUTH2 or SMTP_EXTOPT_BEARER bit flags are set in the options value returned by this method.

Example

```
BOOL bExtended = FALSE;
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server

bExtended = pClient->GetExtendedOptions(&dwOptions);

if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
{
    INT nResult = pClient->Authenticate(lpszUserName, lpszBearerToken,
SMTP_AUTH_XOAUTH2);

    if (nResult == SMTP_ERROR)
    {
        // An error occurred during authentication; when using an
        // OAuth 2.0 bearer token, this typically means that the token
        // has expired and must be refreshed
        return;
    }
}
else
{
    // The server does not support XOAUTH2
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

Authenticate, Connect, GetDeliveryOptions, SetDeliveryOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetLocalName Method

```
INT GetLocalName(
   LPTSTR lpszLocalName,
   INT nMaxLength
);

INT GetLocalName(
   CString& strLocalName
);
```

The **GetLocalName** method returns the local domain name used to identify the client to the mail server.

Parameters

lpszLocalName

The buffer that will contain the local domain name as a string. This may also be a **CString** object which will contain the domain name when the method returns.

nMaxLength

The maximum number of characters that may be copied into the string buffer.

Return Value

If the method succeeds, the return value is the length of the domain name string. If the method fails, the return value is zero, meaning that no local domain name has been specified for this client session.

Remarks

If no local domain name has been explicitly set, then the client will use the default domain name for the local host as configured in the operating system. Note that this is not a general purpose method which can be used to determine the domain name that as been assigned to the local host. It will only return the local domain name set by a previous call to the **SetLocalName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, SetLocalName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

Command, GetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetResultString Method

```
INT GetResultString(
   LPTSTR lpszResult,
   INT cbResult
);

INT GetResultString(
   CString& strResult
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

The buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Command, GetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
SMTP_STATUS_UNUSED	No connection has been established.
SMTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
SMTP_STATUS_CONNECT	The client is establishing a connection with the server.
SMTP_STATUS_READ	The client is reading data from the server.
SMTP_STATUS_WRITE	The client is writing data to the server.
SMTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

IsBlocking, IsConnected, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::GetTransferStatus Method

```
INT GetTransferStatus(
   HCLIENT hClient,
   LPSMTPTRANSFERSTATUS lpStatus
);
```

The **GetTransferStatus** method returns information about the message being submitted to the mail server.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an SMTPTRANSFERSTATUS structure which contains information about the status of the message being submitted for delivery.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current message being submitted, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no message currently being submitted, this method will return the status of the last successful submission made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain the status of a submission for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, GetStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

CSmtpClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
SMTP_EVENT_CONNECT	The connection to the server has completed.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SMTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
SMTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
SMTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **SmtpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a message is being submitted to the server for delivery. The SMTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one SMTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, SmtpEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client cannot be reset while in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, CloseMessage, CreateMessage, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::SendMessage Method

```
INT SendMessage(
   LPCTSTR lpszFrom,
   LPCTSTR lpszRecipient,
   LPCTSTR lpszFileName
);

INT SendMessage(
   LPCTSTR lpszFrom,
   LPCTSTR lpszRecipient,
   LPBYTE lpMessage,
   DWORD dwMessageSize
);

INT SendMessage(
   LPCTSTR lpszFrom,
   LPCTSTR lpszRecipient,
   HGLOBAL hgblMessage,
   DWORD dwMessageSize
);
```

The **SendMessage** method sends the contents of a file or buffer to the specified recipients.

Parameters

lpszFrom

Pointer to a string which specifies the email address of the sender.

lpszRecipient

Pointer to a string which specifies the recipient of the message. Multiple recipients may be specified by separating each address with a comma.

lpszFileName

Pointer to a string which specifies a file that contains the message to be delivered.

lpMessage

Pointer to a buffer which contains the message to be delivered. This may also be a global memory handle which references the message data.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

Return Value

If the method succeeds, the return value is the result code from the server. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method is used to send the contents of a file, buffer or global memory handle to the specified recipients. The message must be in the standard format as described in RFC 822. The **CMailMessage** class can be used to compose and export a message in the correct format.

This protocol is only concerned with the delivery of a message and not its contents. Header fields in the message are not parsed to determine the recipients. This recipient parameter should be a concatenation of all recipients, including carbon copies and blind carbon copies, with each address separated with a comma.

This method will cause the current thread to block until the complete message has been delivered, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of the transaction while it is in progress, use the **GetTransferStatus** method.

An alternative approach to creating a message without using the **CMailMessage** class is the **SubmitMessage** method. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single method call.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AppendMessage, CloseMessage, GetTransferStatus, SubmitMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::SetDeliveryOptions Method

```
BOOL SetDeliveryOptions(
   DWORD dwOptions
);
```

The **SetDeliveryOptions** method sets the delivery status notification options for the current session.

Parameters

dwOptions

A bitmask that defines the current delivery options. This value is created by combining one or more of the following constants with a bitwise Or operator:

Value	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetDeliveryOptions** method sets the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDeliveryOptions, GetExtendedOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::SetLocalName Method

```
BOOL SetLocalName(
   LPCTSTR lpszLocalName
);
```

The **SetLocalName** method sets the local domain name used to identify the client to the mail server.

Parameters

lpszLocalName

A null-terminated string which specifies the local domain name to be used by the client when establishing a connection with the mail server. A value of NULL or an empty string clears the domain name.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetLocalName** method should only be used to specify a domain name if it is absolutely necessary. In most cases, it is preferable to allow the library to automatically determine the correct domain name to use. Providing an invalid domain name may cause the mail server to reject the connection.

This method may only be called prior to establishing a connection with the mail server using the **Connect** method. After the connection has been made, this method will fail. To change the local domain name used by the client, you must first terminate the current session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, GetLocalName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::SubmitMessage Method

```
BOOL SubmitMessage(
   LPSMTPSERVER lpServer,
   LPSMTPMESSAGE lpMessage,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);

BOOL SubmitMessage(
   LPSMTPSERVER lpServer,
   LPSMTPMESSAGEEX lpMessage,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **SubmitMessage** method composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessage

A pointer to an **SMTPMESSAGE** or **SMTPMESSAGEEX** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

lpEventProc

An optional pointer to the procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL or is omitted, event notification is disabled.

dwParam

An optional user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SubmitMessage** method provides a high-level interface that enables an application to send an email message with a single function call. The **SMTPSERVER** and **SMTPMESSAGE** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this method does not require you to call the **Connect** method prior to calling this function.

If you need to specify additional custom headers in the message that is submitted for delivery, you should use the **SMTPMESSAGEEX** structure. It is an extended version of the message structure which

will allow you to define custom headers to be included in the message.

This method will cause the calling thread to block until the message has been submitted for delivery, an error occurs or the connection to the mail server times out. If an event handler is specified, then the callback function will be periodically invoked as the message is being sent. For large messages, the SMTP_EVENT_PROGRESS event can be used to monitor the submission process and update the user interface. The **GetTransferStatus** method can be used within the callback function to obtain information about the current status of the submission.

Example

```
CSmtpClient smtpClient;

SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGE mailMessage;
ZeroMemory(&mailMessage, sizeof(mailMessage));
mailMessage.lpszFrom = m_strSender;
mailMessage.lpszTo = m_strRecipients;
mailMessage.lpszSubject = m_strSubject;
mailMessage.lpszText = m_strMessage;

if (smtpClient.SubmitMessage(&mailServer, &mailMessage))
    _tprintf(_T("SubmitMessage was successful\n"));
else
{
    CString strError;
    smtpClient.GetErrorString(strError);
    _tprintf(_T("SubmitMessage failed: %s\n"), strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SendMessage, SmtpEventProc, SMTPMESSAGE, SMTPMESSAGEEX, SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::VerifyAddress Method

```
INT VerifyAddress(
   LPCTSTR lpszAddress,
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT VerifyAddress(
   LPCTSTR lpszAddress,
   CString& strBuffer
);
```

The **VerifyAddress** method verifies the specified address is valid.

Parameters

lpszAddress

Points to a string which specifies the address that the server should verify.

lpszBuffer

Points to a buffer that the verified address will be copied into. This argument may also be a **CString** object which will contain the verified address when the method returns.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **VerifyAddress** method requests that the server verify the specified email address. Typically this is used to verify that a recipient address is valid, and return a fully qualified email address for that recipient. A server may not support this command, or may restrict its usage. An application should not depend on the ability to verify addresses.

This method cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddRecipient, ExpandAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSmtpClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

If the **Write** method is used to send the message contents to the server, the application must first call the **CreateMessage** method to specify the sender and the length of the message, followed by one or more calls to the **AddRecipient** method to specify each recipient of the message. When all of the message text has been submitted to the server, the application must call the **CloseMessage** method.

The message text is filtered by the **Write** method, and it will automatically normalize end-of-line character sequences to ensure the message meets the protocol requirements. The message itself must be in a standard RFC 822 or multi-part MIME message format, or the server may reject the message. Binary data, such as file attachments, should always be encoded. The **CMailMessage** class can be used to compose and export a message in the correct format, which can then be submitted to the server.

It is recommended that most applications use the **SendMessage** method to submit the message for delivery.

An alternative approach to creating a message without using the **CMailMessage** class is the **SubmitMessage** method. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single method call.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AddRecipient, AppendMessage, CloseMessage, CreateMessage, SendMessage, SubmitMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO
- SMTPMESSAGE
- SMTPMESSAGEEX
- SMTPSERVER
- SMTPTRANSFERSTATUS
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPMESSAGE

This structure provides information about the contents of a message and is used by the SubmitMessage method.

```
typedef struct _SMTPMESSAGE
{
    LPCTSTR  lpszFrom;
    LPCTSTR  lpszTo;
    LPCTSTR  lpszCc;
    LPCTSTR  lpszBcc;
    LPCTSTR  lpszSubject;
    LPCTSTR  lpszText;
    LPCTSTR  lpszHTML;
    LPCTSTR  lpszAttach;
    UINT     nCharSet;
    UINT     nEncType;
    DWORD    dwReserved;
} SMTPMESSAGE, *LPSMTPMESSAGE;
```

Members

lpszFrom

A null-terminated string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A null-terminated string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A null-terminated string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A null-terminated string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the ***lpszTo*** and ***lpszCc*** members, any addresses specified by this member will not be included in the header of the email message.

lpszSubject

A null-terminated string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpszText

A null-terminated string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the ***lpszHTML*** member must specify the body of the message.

lpszHTML

A null-terminated string which contains the message using HTML formatting. If the ***lpszText*** member is not NULL, then a multipart message will be created with both plain text and HTML text

as the alternative. This allows mail clients to select which message body they wish to display. If the ***lpszText*** member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpszAttach

A null-terminated string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

An integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Value	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.
MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be

used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using the **SmtpSubmitMessage** function. It is required that you specify a sender, at least one recipient and a message body. All other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the structure.

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* member cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPMESSAGEEX

This structure provides information about the contents of a message and is used by the SubmitMessage method.

```
typedef struct _SMTPMESSAGEEX
{
    DWORD    dwSize
    LPCTSTR  lpszFrom;
    LPCTSTR  lpszTo;
    LPCTSTR  lpszCc;
    LPCTSTR  lpszBcc;
    LPCTSTR  lpszDate;
    LPCTSTR  lpszSubject;
    LPCTSTR  lpszHeaders;
    LPCTSTR  lpszText;
    LPCTSTR  lpszHTML;
    LPCTSTR  lpszAttach;
    UINT     nCharSet;
    UINT     nEncType;
    DWORD    dwReserved;
} SMTPMESSAGEEX, *LPSMTPMESSAGEEX;
```

Members

dwSize

An integer value that specifies the size of the **SMTPMESSAGEEX** data structure. This must always be explicitly defined, and if the value is incorrect, an error will be returned. This structure member is used to ensure that the correct version of the structure is being passed to the method.

lpszFrom

A null-terminated string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A null-terminated string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A null-terminated string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A null-terminated string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the *lpszTo* and *lpszCc* members, any addresses specified by this member will not be included in the header of the email message.

lpszDate

A null-terminated string that specifies the date and time for the message. This structure member may be NULL or point to an empty string. If the date is not specified, then the current date and time will be used by default. If a date is specified, it should be in the standard format as defined by RFC822.

lpszSubject

A null-terminated string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpszHeaders

A null-terminated string that specifies additional headers that should be included in the message. Header names should be separated from values by a colon, and multiple headers may be defined by separating them with a newline character. This structure member may be NULL, in which case no additional headers will be included in the message.

lpszText

A null-terminated string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the ***lpszHTML*** member must specify the body of the message.

lpszHTML

A null-terminated string which contains the message using HTML formatting. If the ***lpszText*** member is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the ***lpszText*** member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpszAttach

A null-terminated string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

An integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Value	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.
MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the

	application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using the **SubmitMessage** method. It is required that you specify a sender, at least one recipient and a message body. Other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the

structure.

Note that you must explicitly define the size of the structure by setting the value of the **dwSize** member variable. This ensures that the correct version of the structure is being passed to the method.

Email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* member cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

If you specify a message date by assigning a value to the *lpszDate* member, and it does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you provide input from a user, because in most cases they will not include the timezone and will assume the date and time they enter is for their current timezone.

If you wish to include additional headers in the message, you can specify them in a string. Each header consists of a name and value, separated by a colon (":") character. If you wish to define multiple headers, then you can separate them with a newline (e.g.: a linefeed character or combination of a carriage-return and linefeed). Extraneous leading and trailing whitespace are trimmed from header names and values. Invalid names or values will be ignored and will not generate an error.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPSERVER

This structure provides connection information for a mail server and is used by the SubmitMessage method.

```
typedef struct _SMTPSERVER
{
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    UINT     nHostPort;
    UINT     nTimeout;
    DWORD    dwOptions;
    DWORD    dwReserved;
} SMTPSERVER, *LPSMTPSERVER;
```

Members

lpszHostName

A null-terminated string that specifies the host name or IP address of the mail server. This structure member cannot be NULL.

lpszUserName

A null-terminated string that specifies the username that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

lpszPassword

A null-terminated string that specifies the password that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

nHostPort

An integer value that specifies the port number used to establish the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit TLS connections, the default port number is 465.

nTimeout

An integer value that specifies the number of seconds that the client will wait for a response from the server before failing the operation. A value of zero specifies the default timeout period of 20 seconds.

dwOptions

An unsigned integer that specifies one or more options. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not

	specified, the library will not attempt to use any extended features. This option is automatically enabled if a username and password are specified, or if the connection is established on port 587, because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SMTPMESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPTRANSFERSTATUS

This structure is used by the GetTransferStatus method to return information about a message being submitted for delivery.

```
typedef struct _SMTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} SMTPTRANSFERSTATUS, *LPSMTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the size of the message cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Class Library

A general purpose TCP/IP networking library for developing client and server applications.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CSocketWrench
File Name	CSWSKV12.DLL
Version	12.0.3024.1286
LibID	99EEE601-E935-4266-B046-771A823DB47D
Import Library	CSWSKV12.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

At the core of all of the SocketTools networking libraries is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench class library provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that developers face when building Internet applications.

This class library supports secure connections using the TLS 1.2 and TLS 1.3 protocols and can also be used to create secure, customized server applications. Both implicit and explicit TLS connections are supported, enabling the class to work with a wide variety of client and server applications without requiring third-party encryption libraries.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Class Methods

Class	Description
CSocketWrench	Constructor which initializes the current instance of the class
~CSocketWrench	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the connection and immediately close the socket
Accept	Accept a connection request from a remote host
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified socket to another thread
Cancel	Cancel a blocking operation
CompareAddress	Compare two IP addresses to determine if they are identical
Connect	Connect to the specified server
ConnectUrl	Connect to the specified server using a URL
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableSecurity	Disable secure communication with the remote host
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableSecurity	Enable secure communication with the remote host
EnableTrace	Enable logging of network function calls to a file
EnumNetworkAddresses	Return the list of network addresses that are configured for the local host
Flush	Flush the send and receive buffers
FormatAddress	Convert an IP address in binary format into a printable string
FreezeEvents	Suspend or resume event handling by the application
GetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
GetAddress	Convert an IP address string to a binary format
GetAddressFamily	Return the address family for the specified IP address
GetDefaultHostFile	Return the fully qualified path name of the host file on the local system
GetErrorString	Return a description for the specified error code
GetExternalAddress	Return the external IP address assigned to the local system

GetFirstAlias	Return the first alias for the specified host name
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the IP address assigned to the specified hostname
GetHostFile	Return the name of the host file
GetHostName	Return the hostname assigned to the specified IP address
GetLastError	Return the last error code
GetLocalAddress	Return the local IP address and port number for a socket
GetLocalName	Return the hostname assigned to the local system
GetMemoryUsage	Return the amount of memory allocated for the network connection
GetNextAlias	Return the next alias for the specified host name
GetOption	Return the current socket options
GetPeerAddress	Return the IP address of the peer that the socket is connected to
GetPeerPort	Returns the remote port number used by the client to establish the connection
GetPhysicalAddress	Return the media access control (MAC) address for the primary network adapter
GetSecurityInformation	Return information about the security characteristics of a connection
GetServiceName	Return the service name associated with a specified port number
GetServicePort	Return the port number associated with a service name
GetStatus	Report what sort of socket operation is in progress
GetStreamInfo	Return information about the current stream read or write operation
GetTimeout	Return the timeout interval for blocking operations, in seconds
GetUrlHostName	Return the host name and port number specified in a URL
GetVpnAdapter	Return the name of the active Virtual Private Network adapter
HostNameToUnicode	Converts the canonical form of a host name to its Unicode version
InetEventProc	Callback method that processes events generated on the socket
IsAddressNull	Determine if the specified IP address is a null address
IsAddressRoutable	Determine if the specified IP address is routable over the Internet
IsBlocking	Determine if the socket is performing a blocking operation
IsClosed	Determine if the remote host has closed its socket
IsConnected	Determine if the socket is connected to a remote host
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the socket is listening for a connection
IsProtocolAvailable	Determine if the specified protocol and address family are supported
IsReadable	Determine if data can read from the socket without blocking
IsUrgent	Determine if there is any out-of-band data available to be read
IsWritable	Determine if data can be written to to the socket without blocking

Listen	Listen for client connections on the specified socket
MatchHostName	Match a host name against of list of addresses including wildcards
NormalizeHostName	Return the canonical form of a host name
Peek	Read data from the socket without removing it from the socket buffer
Read	Read data from the socket
ReadLine	Read a line of data from the socket, storing it in a string buffer
ReadStream	Read a stream of data from the socket
RegisterEvent	Register an event callback function
Reject	Reject a pending client connection
SetHostFile	Specify the name of an alternate host table
SetLastError	Set the last error code
SetOption	Set one or more options for the current socket
SetTimeout	Set the interval used when waiting for a blocking operation to complete
ShowError	Display a message box with a description of the specified error
Shutdown	Disable reception or transmission of data
StoreStream	Read a stream of data from the socket and store it in a file
ValidateCertificate	Validate the specified security certificate is installed on the local system
ValidateHostName	Validate the specified host name and return the resolved IP address
Write	Write data to the socket
WriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the socket

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::CSocketWrench Method

```
CSocketWrench();
```

The **CSocketWrench** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **InetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

~CSocketWrench, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::~CSocketWrench

```
~CSocketWrench();
```

The **CSocketWrench** destructor releases resources allocated by the current instance of the **CSocketWrench** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CSocketWrench** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the connection are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CSocketWrench

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Abort Method

```
BOOL Abort();
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Abort** method should only be used when the connection must be closed immediately before the application terminates. This method should only be used to abort client connections and should not be used with passive (listening) sockets. Server applications that need to abort an incoming client connection should use the **Reject** method.

In most cases, the application should call the **Disconnect** method to gracefully close the connection to the remote host. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Cancel, Disconnect, Reject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Accept Method

```
BOOL Accept(
   SOCKET hSocket,
   UINT nTimeout,
   DWORD dwOptions
);

BOOL Accept(
   CSocketWrench& swServer,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Accept** method is used to accept a pending client connection.

This method has been deprecated and is included for backwards compatibility. Use the **CInternetServer** class to create a server application.

Parameters

hSocket

Handle to the listening socket. This argument may also reference a **CSocketWrench** object which is listening for connections. In either case, the server socket must have been created by calling the **Listen** method.

nTimeout

The number of seconds that the server will wait for a client connection before failing the operation. This value is used only for blocking connections.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle,

	leaving it open after the parent process has stopped the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option determines if a secure connection is established with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. If no event notification window is specified, then **Accept** will block until a client attempts to connect to the server or the timeout period expires.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Accept, Connect, Disconnect, Listen

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::AttachHandle Method

```
VOID AttachHandle(
   SOCKET hSocket
);

VOID AttachHandle(
   SOCKET hSocket,
   DWORD dwProcessId
);
```

The **AttachHandle** method attaches the specified socket handle to the current instance of the class.

Parameters

hSocket

The socket handle that will be attached to the current instance of the class object.

dwProcessId

The process ID for the process that currently owns the socket handle. This value may be zero to specify the current process.

Return Value

None.

Remarks

This method is used to attach a socket handle created outside of the class using the SocketWrench API. Once the socket handle is attached to the class, the other class member functions may be used with that socket. If the socket was created by a third-party library or the Windows Sockets API, then the handle will be automatically inherited by the library.

If a socket handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous socket connection from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

If the *dwProcessId* parameter specifies another process, the socket will be duplicated into the current process, attached to the current thread and the original socket handle will be closed in the other process. This enables an application to effectively take control of a connection created by another process. The original socket handle must be inheritable by the by the current process and must be an actual Windows socket handle, not a pseudo-handle. This functionality is only supported on Windows NT 4.0 and later versions of the operating system with the Microsoft TCP/IP stack. Note that Layered Service Providers (LSPs) may interfere with the ability to inherit handles across processes.

If the socket was created by another process, it is initialized by the library in a blocking state, even if was originally using asynchronous socket events. If the application requires that the socket use events, it must explicitly call **EnableEvents**. A program should never try to attach to a secure connection created by another process because the attached socket will not have the security context required to encrypt and decrypt the data exchanged with the remote host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Cancel Method

```
BOOL Cancel(
   SOCKET hSocket
);

BOOL Cancel();
```

The **Cancel** method cancels any outstanding blocking socket operation, causing the blocking method to fail. The application may then retry the operation or terminate the connection.

Parameters

hSocket

An optional parameter that specifies the handle to the socket. If this parameter is omitted, the socket handle for the current class instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This method is typically called from within an event handler to signal that the current blocking operation should stop. It may also be used to cancel a blocking operation that is occurring on another thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Abort, Disconnect, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::CompareAddress Method

```
BOOL CompareAddress(
   LPINTERNET_ADDRESS lpAddress1,
   LPINTERNET_ADDRESS lpAddress2
);

BOOL CompareAddress(
   LPCTSTR lpszAddress1,
   LPCTSTR lpszAddress2
);
```

Compare two IP addresses to determine if they are identical.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

Return Value

If the method succeeds and the two addresses are identical, the return value is non-zero. If the method fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetHostAddress, GetLocalAddress, GetPeerAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort
);

BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nProtocol,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort
);
```

The **Connect** method is used to establish a connection with a server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the system you want to connect with. This parameter cannot be a URL and must only specify the name of the remote host. If this parameter is NULL or an empty string, the method will fail with an error indicating the host name is invalid.

nRemotePort

The port number used to establish the connection. Valid port numbers range in value from 1 through 65535 and a value outside if this range will cause the function to fail. Port numbers in the range of 49152 and 65535 are referred to as dynamic ports and are generally reserved for private use by client applications. You cannot specify a port number of zero when establishing an outbound connection.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description

INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the ***lpCredentials*** parameter. If the ***lpCredentials*** parameter is NULL, the connection will default to using TLS 1.2 or later and the strongest cipher suites available.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a

	connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

lpszLocalAddress

A null-terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The *lpszHostName* parameter must specify a valid host name or IP address. Host names are resolved into an IP address by first checking the local hosts file and if the name is not found, a name server query will be performed to determine the IP address. If the Unicode version of this function is called and the host name includes non-ASCII characters, the host name will be automatically converted to an ASCII compatible format. Refer to the **NormalizeHostName** method for more information. To establish a connection using a URL rather than a host name, use the **ConnectUrl** method.

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **EnableSecurity** method when you are ready to initiate the TLS handshake.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option

disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, Disconnect, EnableSecurity, Read, ReadLine, RegisterEvent, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ConnectUrl Method

```
BOOL ConnectUrl(
   LPCTSTR lpszUrl,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **ConnectUrl** method is used to establish a TCP connection with a server using the information provided in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL used when establishing the connection. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the method will fail. See the remarks below for more information on the format supported by this method.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation. This parameter is optional and if omitted or the value is zero, a default timeout period will be used.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is optional and if omitted, no additional options will be specified. This parameter value is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is not necessary for most connections, particularly when the client will not be connected to the server for an extended period of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will

	always be permitted. This option only affects secure connections using the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The connection will always default to using TLS 1.2 or later and the strongest cipher suites available on the client platform. This option may be automatically enabled if the URL scheme specifies a service which requires a secure connection. See the remarks below for more information.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ConnectUrl** method provides a simplified interface which can be used to establish a connection using a URL. This method can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters
...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme

would be used, which is port 443. The host name **www.example.com** would be resolved into an IP address and the connection established on port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the method cannot determine what port number should be used when establishing the connection. The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this method will automatically enable the INET_OPTION_SECURE option. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection regardless if the *dwOptions* parameter includes that option. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

```
http://www.example.com:443
```

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if INET_OPTION_SECURE has not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, this method will return an error indicating the port number is invalid. For example, the URL `https://[2001:db8:0:0:1::128]/` uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this method will only be used to establish a connection with a server. This is a general purpose method which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools API provides comprehensive collection of higher-level classes which can be used to access those services.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **InetEnableSecurity** method when you are ready to initiate the TLS handshake.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnectUrl** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that handle. The ownership of the handle may be transferred from one thread to another using the

InetAttachThread method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a method in the library, it must ensure that no other thread will call another method at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect, EnableSecurity, GetUrlHostName, Read, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pSocket->CreateSecurityCredentials(lpszCertName);

bConnected = pSocket->Connect(lpszHostName,
                              INET_PORT_HTTP,
                              INET_PROTOCOL_TCP,
                              INET_TIMEOUT,
                              INET_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, ValidateCertificate, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current connection.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials, ValidateCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::DetachHandle Method

```
SOCKET DetachHandle();
```

The **DetachHandle** method detaches the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active connection, the value INVALID_SOCKET will be returned.

Remarks

This method is used to detach a socket handle created by the class for use with the SocketWrench API. Once the socket handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly closed at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::DisableEvents Method

```
BOOL DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified socket handle. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the socket has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a socket handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::DisableSecurity Method

```
INT DisableSecurity();
```

The **DisableSecurity** method disables a secure session with the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableSecurity** method disables a secure session, with subsequent calls to **Read** and **Write** sending and receiving unencrypted data. It is important to note that because this method sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This method does not close the socket. Use the **Disconnect** method to close the socket and release the resources allocated for the current session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CreateSecurityCredentials, DeleteSecurityCredentials, EnableSecurity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Disconnect Method

```
BOOL Disconnect();
```

Terminate the connection, closing the socket and releasing the memory allocated for the session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Once the connection has been terminated, the class instance socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Abort, Accept, Connect, DisableEvents, EnableEvents, Listen

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::EnableEvents Method

```
BOOL EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the socket notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a network event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a network event occurs. This allows an application to monitor the status of different socket operations.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection to the client. The client should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to be read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
INET_EVENT_WRITE	The application can now send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a

	blocking operation. This event is only generated if the client is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::EnableSecurity Method

```
BOOL EnableSecurity();

BOOL EnableSecurity(
   LPCTSTR lpszCertName
);

BOOL EnableSecurity(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);
```

The **EnableSecurity** method enables a secure session with the remote host.

Parameters

lpszCertStore

A null-terminated string which specifies the name of certificate store.

lpszCertName

A null-terminated string which specifies the common name for the certificate that will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableSecurity** method enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate. If the socket was created using the **Connect** method to establish a client connection, then **EnableSecurity** will initiate the handshake with the remote host to establish a secure session. If the **Accept** method was used to accept a connection from a client, then the method will block and wait for the remote host to initiate the handshake.

This method is useful if the application needs to establish an initial, non-secure connection to the remote host and then negotiate a secure connection at a later point. If the method succeeds, all subsequent calls to **Read** and **Write** to receive and send data will be encrypted.

If no arguments are specified, then the security credentials established with a previous call to **CreateSecurityCredentials** will be used. If a certificate name is specified, then the current security credentials will be updated to use that certificate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CreateSecurityCredentials, DeleteSecurityCredentials, DisableSecurity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::EnumNetworkAddresses Method

```
INT EnumNetworkAddresses(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddressList,
   INT nMaxAddresses
);

INT EnumNetworkAddresses(
   LPINTERNET_ADDRESS lpAddressList,
   INT nMaxAddresses
);
```

The **EnumNetworkAddresses** method returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this method. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses assigned to the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that require support for IPv6 connections.
INET_ADDRESS_IPV4	Return only the IPv4 addresses assigned to the local host. The first four bytes of the *ipNumber* array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Return only the IPv6 addresses assigned to the local host. All bytes in the *ipNumber* array are significant. This option is only recommended for those applications that require support for IPv6 connections.

lpAddressList

A pointer to an array of INTERNET_ADDRESS structures that will contain the IP address of each local network interface. This parameter may be NULL, in which case the method will only return the number of available addresses.

nMaxAddresses

Maximum number of addresses to be returned. If the *lpAddressList* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the number of network addresses that are configured for the local host. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to accept IPv6 addresses returned by this method. On Windows Vista and later versions of the operating system, IPv6 support is enabled and the local network adapter will have IPv6 addresses assigned to them by default. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* parameter should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

This method will ignore addresses that are bound to a disabled interface, as well as those addresses bound to a virtual loopback interface. For example, although the loopback address 127.0.0.1 is a valid network address, it will not be included in list of addresses returned by this method.

The first IPv4 or IPv6 address returned by this method is typically the address assigned to the primary network adapter on the local system. However, your application should not depend on addresses being returned in any particular order. If the system has virtualization software installed, this method may also include the IP addresses assigned to any virtualized network adapters installed by that software.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAdapterAddress, GetHostAddress, GetLocalAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Flush Method

```
BOOL Flush();
```

The **Flush** method flushes the internal send and receive buffers used by the socket.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** method or prior to calling the **Disconnect** method.

An application never needs to use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::FormatAddress Method

```
INT FormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszAddress,
   INT cchAddress
);

INT FormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   CString& strAddress
);
```

The **FormatAddress** method converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

The buffer that will contain the formatted IP address. This buffer should be at least 46 characters in length. This may also reference a **CString** object which will contain the formatted address when the method returns.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the IP address string. If the method fails, the return value is INET_ERROR, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the INTERNET_ADDRESS structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetLocalAddress, GetPeerAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the application.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all socket events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the program has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetAdapterAddress Method

```
INT GetAdapterAddress(
   INT nAdapterIndex,
   INT nAddressType,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetAdapterAddress(
   INT nAdapterIndex,
   INT nAddressType,
   CString& strAddress
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Value	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type. An alternate form of the method accepts a **CString** argument which will contain the hostname.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the method fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **GetLastError**.

Remarks

The **GetAdapterAddress** method will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this method would return zero.

This method will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this method may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    CString strAddress;

    if (pSocket->GetAdapterAddress(nIndex, INET_ADAPTER_IPV4, strAddress) ==
INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnumNetworkAddresses, GetLocalAddress, GetLocalName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetAddress Method

```
INT GetAddress(
   LPCTSTR lpszAddress,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);

INT GetAddress(
   LPCTSTR lpszAddress,
   LPINTERNET_ADDRESS lpAddress
);
```

The **GetAddress** method converts an IP address string to binary format.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the ***lpszAddress*** parameter. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the ***lpszAddress*** parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the ***lpszAddress*** parameter does not specify a valid IPv4 address string, this method will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the ***ipNumber*** array are significant. If the ***lpszAddress*** parameter does not specify a valid IPv6 address string, this method will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member

should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the ***lpszAddress*** parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, IsAddressNull, IsAddressRoutable, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetAddressFamily Method

```
INT GetAddressFamily(
   LPCTSTR lpszAddress,
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the method succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the method fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call the **GetLastError** method.

Value	Description
INET_ADDRESS_IPV4	The address passed to the method is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the method is a valid IPv6 address.

Remarks

The **GetAddressFamily** method returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other methods such as **Connect**. Note that this method will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsAddressNull, IsAddressRoutable, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetDefaultHostFile Method

```
INT GetDefaultHostFile(
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetDefaultHostFile(
   CString& lpszFileName
);
```

The **GetDefaultHostFile** method returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The location of the default host file depends on the operating system. For Windows 95/98 and Windows Me the file is stored in C:\Windows\hosts and for Windows NT and later versions the file is stored in C:\Windows\system32\drivers\etc\hosts. Regardless of platform, there is no filename extension and this file may or may not exist on a given system.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **GetLastError**.

Remarks

This method only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **SetHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostFile, GetHostName, SetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetExternalAddress Method

```
INT GetExternalAddress(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress,
   INT nMaxLength
);

INT GetExternalAddress(
   INT nAddressFamily,
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetExternalAddress(
   INT nAddressFamily,
   CString& strAddress
);
```

The **GetExternalAddress** method returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return any external IP address available. IPv4 addresses will be preferred over IPv6 for backwards compatibilty.
INET_ADDRESS_IPV4	Specifies that the external address should be in IPv4 format.
INET_ADDRESS_IPV6	Specifies that the external address should be in IPv6 format. If the local system does not have IPv6 enabled, the method will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the external IP address of the local host in binary form.

lpszAddress

A null-terminated string buffer that will contain the external IP address of the local host.

nMaxLength

The maximum length of the string that will contain the IP address when the method returns.

Return Value

If the method succeeds, the return value is zero. If the function fails, it will return INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **GetExternalAddress** method returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **GetLocalAddress** method will only return the IP address for the local system on the

LAN side of the network unless a connection has already been established to a remote host. The **GetExternalAddress** method can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

An application should always check the return value in case there is an error. Never assume that the return value is always a valid address. The method may be unable to determine the external IP address for the local system for a number of reasons, particularly if it is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the method is able to obtain a valid external address for the local host, that address will be cached by the library. If you are using a VPN (Virtual Private Network), this method will return the external IP address for the VPN adapter and not your local network connection. The **GetVpnAdapter** method can be used to obtain information about the active VPN adapter.

Calling this method may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this method in an application which uses asynchronous sockets, it is recommended that you create a background worker thread and call this method from within that thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, GetHostAddress, GetLocalAddress, GetPeerAddress, GetVpnAdapter, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetFirstAlias Method

```
BOOL GetFirstAlias(
   LPCTSTR lpszHostName,
   LPTSTR lpszHostAlias,
   INT nMaxLength
);

BOOL GetFirstAlias(
   LPCTSTR lpszHostName,
   CString& strHostAlias
);
```

The **GetFirstAlias** method returns the first alias for the specified host name.

Parameters

lpszHostName

A null-terminated string which specifies the host name that you wish to return aliases for. This should be complete domain name.

lpszHostAlias

A string buffer which will contain the first alias for the specified host name. This string should be at least 64 bytes in length. This argument may also reference a **CString** object which will contain the host alias when the method returns.

nMaxLength

Maximum number of characters that can be copied into the *lpszHostAlias* string buffer, including the terminating null byte.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
CSocketWrench sockClient;
CString strHostAlias;
BOOL bResult;

m_ctlListBox.ResetContent();

bResult = sockClient.GetFirstAlias(m_strHostName, strHostAlias);
if (bResult == FALSE)
{
    sockClient.ShowError();
    return;
}

while (bResult)
{
    m_ctlListBox.AddString(strHostAlias);
    bResult = sockClient.GetNextAlias(strHostAlias);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, GetNextAlias

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetHandle Method

```
SOCKET GetHandle();
```

The **GetHandle** method returns the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the socket handle associated with the current instance of the class object. If there is no active connection, the value INVALID_SOCKET will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetHostAddress Method

```
INT GetHostAddress(
   LPCTSTR lpszHostName,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);

INT GetHostAddress(
   LPCTSTR lpszHostName,
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetGetHostAddress** method resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

The name of the host to resolve; this may be a fully-qualified domain name or an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the *ipNumber* array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the *ipNumber* array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the specified host.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This method can also be used to convert an address in dot notation to a binary format. If the method must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future

compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the ***ipFamily*** member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The ***nAddressFamily*** parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the ***lpszHostName*** parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostName, GetLocalAddress, GetLocalName, GetPeerAddress, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetHostFile Method

```
INT GetHostFile(
   LPTSTR lpszFileName,
   INT nMaxLength
);

INT GetHostFile(
   CString& strFileName
);
```

The **GetHostFile** method returns the name of the host file previously set using the **SetHostFile** method. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the method was unable to determine the file name. To get extended error information, call **GetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the method failed.

Remarks

This method only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **SetHostFile** method.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this method may be different than the default host file for the local system. To determine the file name for the default host file, use the **GetDefaultHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetHostAddress, GetHostName, SetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetHostName Method

```
INT GetHostName(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszHostName,
   INT cchHostName
);

INT GetHostName(
   LPINTERNET_ADDRESS lpAddress,
   CString& strHostName
);
```

The **GetHostName** method performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the IP address that should be resolved into a host name.

lpszHostName

The buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the method must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This method requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this method frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetLocalAddress, GetLocalName, GetPeerAddress, IsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetLocalAddress Method

```
INT GetLocalAddress(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnPort
);

INT GetLocalAddress(
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetLocalAddress(
   CString& strAddress,
   UINT *lpnPort
);
```

The **GetLocalAddress** method returns the local IP address and port number for the current socket.

Parameters

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the type of connection that was established. If the ***hSocket*** parameter is INVALID_SOCKET, a preference will be given for returning an IPv4 address. However, if the local host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the local host. If there is no active connection, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpszAddress

A null-terminated string that will contain the IP address of the local host. If this version of the method is used, the IP address is converted to a string format using the **FormatAddress** method. The string should be able to store at least 46 characters to ensure that both IPv4 and IPv6

formatted addresses can be returned without the possibility of a buffer overrun. An alternate form of the method accepts a **CString** argument which will contain the local address.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If there is an active connection, this parameter will be set to the local port that the socket is bound to. If there is no active connection, this parameter is ignored. If the local port number is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the ***ipFamily*** member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for the local host to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **GetLocalAddress** method will return the local, non-routable IP address assigned to the local system. To determine the public IP address has been assigned to the system, you should use the **GetExternalAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetExternalAddress, GetHostAddress, GetHostName, GetLocalName, GetPeerAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetLocalName Method

```
INT GetLocalName(
   LPTSTR lpszHostName,
   INT cchHostName
);

INT GetLocalName(
   CString& strHostName
);
```

The **GetLocalName** method returns the hostname assigned to the local system.

Parameters

lpszHostName

The buffer that will contain the hostname. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the local hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, GetLocalAddress, GetPeerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the network connection.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of bytes allocated. If the method fails, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated for the socket. It enumerates all heap allocations made by the process and returns the total number of bytes allocated for the current socket handle. This value only reflects the amount of memory explicitly allocated by this class library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the current thread into a locked state, and all network operations will block until the method returns. Because this method enumerates all heaps created for the process, it can be an expensive operation, particularly when a large number of connections have been established. Frequent use of this method can significantly degrade the performance of the application. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active network connections. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetNextAlias Method

```
BOOL GetNextAlias(
   LPTSTR lpszHostAlias,
   INT nMaxLength
);

BOOL GetNextAlias(
   CString& strHostAlias
);
```

The **GetNextAlias** method returns the next alias for the host name specified in the call to **GetFirstAlias**.

Parameters

lpszHostAlias

A string buffer which will contain the next alias for the specified host name. This string should be at least 64 bytes in length. This argument may also reference a **CString** object which will contain the host alias when the method returns.

nMaxLength

Maximum number of characters that can be copied into the *lpszHostAlias* string buffer, including the terminating null byte.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
CSocketWrench sockClient;
CString strHostAlias;
BOOL bResult;

m_ctlListBox.ResetContent();

bResult = sockClient.GetFirstAlias(m_strHostName, strHostAlias);
if (bResult == FALSE)
{
    sockClient.ShowError();
    return;
}

while (bResult)
{
    m_ctlListBox.AddString(strHostAlias);
    bResult = sockClient.GetNextAlias(strHostAlias);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstAlias, GetHostAddress, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetOption Method

```
INT GetOption(
   DWORD dwOption,
   LPBOOL lpbEnabled
);
```

The **GetOption** method is used to determine if a specific socket option has been enabled.

Parameters

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

lpbEnabled

A pointer to a boolean flag. If the option is enabled, the flag is set to a non-zero value, otherwise it is set to a value of zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Connect, SetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetPeerAddress Method

```
INT GetPeerAddress(
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnRemotePort
);

INT GetPeerAddress(
   LPTSTR lpszAddress,
   INT nMaxLength
);

INT GetPeerAddress(
   CString& strAddress
);
```

The **GetPeerAddress** method returns the peer IP address and remote port number for the specified socket.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the remote host that is connected to the socket.

lpnRemotePort

A pointer to an unsigned integer that will contain the port number of the remote host that is connected to the socket.

lpszAddress

A null-terminated string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

nMaxLength

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If this method is called by a server application in response to a INET_EVENT_ACCEPT event, it will return the IP address and port number for the client that is attempting to establish the connection. If the peer address is unavailable, the ***ipFamily*** member of the INTERNET_ADDRESS structure will be zero.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

It is not recommended that you use the port number for anything other than informational and logging purposes. Server applications should not make any assumptions about the specific port

number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetHostAddress, GetHostName, GetLocalAddress, GetLocalName, GetPeerPort, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetPhysicalAddress Method

```
BOOL GetPhysicalAddress(
   LPTSTR lpszAddress,
   UINT cchAddress
);

BOOL GetPhysicalAddress(
   CString& strAddress
);
```

Return the media access control (MAC) address for the primary network adapter.

Parameters

lpszAddress

A string buffer that will contain the address in a printable format when the function returns. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the address.

cchAddress

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetPhysicalAddress** method returns the media access control (MAC) address for the primary network adapter. This is a 48 bit or 64 bit address that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address as a printable string, with each byte of the address as a two-digit hexadecimal value separated by a colon. The string buffer passed to the method should be at least 20 characters long to accommodate the address and terminating null character. An example of a 48 bit address would be "01:23:45:67:89:AB". If the local system is multi-homed (having more than one network adapter) then this method will return the MAC address for the primary network adapter.

This method is provided for backwards compatibility with previous versions of the library and it is recommended that new applications use the **GetAdapterAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumNetworkAddresses, GetAdapterAddress, GetLocalName, GetHostAddress

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pSocket->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, EnableSecurity, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetServiceName Method

```
BOOL GetServiceName(
   UINT nServicePort,
   LPTSTR lpszServiceName,
   INT nMaxLength
);

BOOL GetServiceName(
   UINT nServicePort,
   CString& strServiceName
);
```

The **GetServiceName** method returns the service name associated with a specified port number.

Parameters

nServicePort

Port number associated with some network service.

lpszServiceName

A null-terminated string buffer that will contain the service name when the method returns. This may also reference a **CString** object that will contain the service name.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetServicePort Method

```
UINT GetServicePort(
   LPCTSTR lpszServiceName
);
```

The **GetServicePort** method returns the port number associated with a service name.

Parameters

lpszServiceName

A null-terminated string which specifies the name of the service to return the port number for.

Return Value

If the method succeeds, the return value is the port number associated with a service name. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetServiceName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method returns the current status of the socket.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value is one of the following values:

Value	Description
INET_STATUS_UNUSED	No connection has been established.
INET_STATUS_IDLE	The socket is idle and not in a blocked state
INET_STATUS_LISTEN	The socket is listening for inbound connections from a client
INET_STATUS_CONNECT	The socket is establishing a connection with a server
INET_STATUS_ACCEPT	The socket is accepting a connection from a client
INET_STATUS_READ	Data is being read from the socket
INET_STATUS_WRITE	Data is being written to the socket
INET_STATUS_FLUSH	The socket is being flushed; all data in the receive buffers is being discarded
INET_STATUS_DISCONNECT	The socket is disconnecting from the remote host

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsConnected, IsInitialized, IsListening, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetStreamInfo Method

```
BOOL GetStreamInfo(
   LPINETSTREAMINFO lpStreamInfo
);
```

The **GetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

lpSecurityInfo

A pointer to an INETSTREAMINFO structure which contains information about the status of the network stream.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStreamInfo** method returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this method will return the status of the last successful streaming read or write performed by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

ReadStream, StoreStream, WriteStream, INETSTREAMINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetUrlHostName Method

```
INT GetUrlHostName(
   LPCTSTR lpszUrl,
   LPTSTR lpszHostName,
   INT nMaxLength,
   UINT *lpnHostPort,
   UINT *lpnProtocol,
   DWORD *lpdwOptions
);

INT GetUrlHostName(
   LPCTSTR lpszUrl,
   CString& strHostName,
   UINT *lpnHostPort,
   UINT *lpnProtocol,
   DWORD *lpdwOptions
);
```

The **GetUrlHostName** method returns the host name and port number specified in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the method will fail. See the remarks below for more information on the format supported by this method.

lpszHostName

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

The maximum number of characters that can be copied to the ***lpszHostName*** string buffer. This parameter cannot be zero, and must include the terminating null character.

lpnHostPort

Pointer to an optional unsigned integer value which will contain the port number specified in the URL. This parameter value will always be initialized by the method with a value of zero. If this parameter is omitted or NULL, it will be ignored and no port information will be returned.

lpnProtocol

Pointer to an optional unsigned integer value which will contain the protocol associated with the URI scheme. This parameter value will always be initialized by the method with a value of zero. If this parameter is omitted or NULL, it will be ignored and no protocol information will be returned.

lpdwOptions

Pointer to an optional unsigned integer value which will contain any socket options required to establish a connection based on the URI scheme or specified port. This parameter value will always be initialized by the method with a value of zero. If this parameter is omitted or NULL, it will be ignored.

Return Value

If the method succeeds, the return value is the number of characters copied into the *lpszHostName* buffer. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **GetUrlHostName** method will extract the host name and port number from a URL and return the canonical form of the host name. If the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters have been specified, they will contain the port number, protocol and additional connection options associated with the URL scheme.

The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters
...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com/
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

If the *lpszUrl* parameter only specifies a host name without a URI scheme or port number, this method will ignore the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters and return the canonical form of the host name in the *lpszHostName* string argument.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, the method will return an error indicating the port number is invalid. For example, the URL `https://[2001:db8:0:0:1::128]/` uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

If the URL uses an IP address instead of a host name, this method will return a copy of that IP address in the *lpszHostName* string provided by the caller. The method will not attempt to resolve the IP address into a host name, however you can use the **GetHostName** method to perform a reverse DNS lookup if required.

The only URI schemes currently supported by this method use TCP stream connections. In practical terms, this means the *lpnProtocol* parameter will always return with the value INET_PROTOCOL_TCP when the method is successful. If the method fails, this value will be INET_PROTOCOL_NONE.

Although this method performs checks to ensure that the *lpszUrl* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the host name. To check if the host name exists and has a valid IP address, use the **ValidateHostName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConnectUrl, GetHostAddress, GetHostName, HostNameToUnicode, NormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetVpnAdapter Method

```
BOOL GetVpnAdapter(
   LPTSTR lpszAdapterName,
   INT nMaxLength,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);

BOOL GetVpnAdapter(
   CString &strAdapterName,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **GetVpnAdapter** method returns the name of the active VPN adapter.

Parameters

lpszAdapterName

A null-terminated string buffer which will contain the name of the active VPN adapter if one is found. If a VPN was not detected, the buffer will contain an empty string when the method returns. This parameter may be NULL if the adapter name is not required. An alternate version of this method accepts a reference to a **CString** object which will contain the adapter name when the method returns.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. If this value is zero, the ***lpszAdapterName*** parameter is ignored.

nAddressFamily

An optional integer value which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return the IP address for the adapter in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the adapter only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the adapter address should be returned in IPv4 format. The first four bytes of the ***ipNumber*** member of the INTERNET_ADDRESS structure are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the adapter address should be returned in IPv6 format. All bytes in the ***ipNumber*** member of the INTERNET_ADDRESS structure are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address assigned to the VPN adapter when the function returns. This parameter may be NULL or omitted if this information is not required. If this parameter is NULL, the value of the ***nAddressFamily*** parameter is ignored.

Return Value

If the method succeeds, the return value is non-zero if an active VPN adapter was found. If no VPN adapter was detected, this method will return zero.

Remarks

The **GetVpnAdapter** method provides a best-effort method to detect if the system is currently using a Virtual Private Network (VPN) connection. The detection logic is based on identifying active network adapters with names or descriptions that match known VPN software or common VPN adapter naming conventions, while explicitly excluding adapters associated with virtualization platforms (such as Hyper-V, VMware, and VirtualBox), loopback interfaces, and other non-VPN virtual adapters.

Because there is no industry standard for how VPN clients are identified at the network adapter level, this method uses a heuristic approach and cannot guarantee detection in every scenario. Some VPN clients may use custom adapter names or attempt to obscure their presence, and new VPN software may not be recognized until the internal detection list is updated. There is also a small chance that non-VPN software using similar naming conventions could result in a false positive.

The result of this method should be treated as a hint for diagnostic or troubleshooting purposes. It should not be used to restrict access to your application or deny service to users based solely on the possible presence of a VPN connection. If you specify an address family and the ***lpAddress*** parameter is not NULL, it will contain the first IP address bound to the adapter interface. It is possible for VPN adapters to be bound to multiple IP addresses. The **FormatAddress** method can be used to convert the INTERNET_ADDRESS value to a printable string.

The adapter name returned by this method is the description of the detected VPN adapter, encoded as a UTF-8 string in the ANSI version of this function. If you require more detailed network information, consider using the **EnumNetworkAddresses** method. You can use the **GetExternalAddress** method to obtain the public-facing IP address allocated for the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumNetworkAddresses, FormatAddress, GetAdapterAddress, GetExternalAddress, GetLocalAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::HostNameToUnicode Method

```
INT HostNameToUnicode(
   LPCTSTR lpszHostName,
   LPTSTR lpszUnicodeName,
   INT nMaxLength
);

INT HostNameToUnicode(
   LPCTSTR lpszHostName,
   CString& strUnicodeName
);
```

The **HostNameToUnicode** method converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **HostNameToUnicode** method will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this method, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the application is compiled using the Unicode character set, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI character set is used, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the GetHostAddress method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, NormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsAddressNull Method

```
BOOL IsAddressNull(
   LPCTSTR lpszAddress
);

BOOL IsAddressNull(
   LPINTERNET_ADDRESS lpAddress
);
```

The **IsAddressNull** method determines if the IP address is null.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is null, or the parameter is a NULL pointer, the return value is non-zero. If the method fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAddress, IsAddressRoutable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsAddressRoutable Method

```
BOOL IsAddressRoutable(
   LPCTSTR lpszAddress
);

BOOL IsAddressRoutable(
   LPINTERNET_ADDRESS lpAddress
);
```

The **IsAddressRoutable** method determines if the IP address is routable over the Internet.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is routable over the Internet, the return value is non-zero. If the method fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAddress, GetExternalAddress, IsAddressNull, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the socket is performing a blocking operation.

Parameters

None.

Return Value

If the socket is currently performing a blocking operation, the method returns a non-zero value. If the socket is not performing a blocking operation, or the socket handle is invalid, the method returns zero.

Remarks

This method is typically used to determine if a socket that is being used by another thread is currently blocked. A socket may block when waiting to receive data from a remote host or while data is actively being exchanged. Because there can only be one blocking socket operation per thread, this method can be used to determine if a method such as **Read** or **Write** would fail because another thread is currently sending or receiving data on that socket.

It is important to note that if this method returns a non-zero value, it does not guarantee that a subsequent read or write on the socket will succeed. The application should always check the return value from methods such as **Read** and **Write** to ensure they were successful.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsConnected, IsReadable, IsWritable, Read, ReadLine, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsClosed Method

```
BOOL IsClosed();
```

The **IsClosed** method is used to determine if the remote host has closed its socket.

Parameters

None.

Return Value

If the remote host has closed its socket, the method returns a non-zero value. If the remote host has not closed its connection, or the socket handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsConnected, IsListening, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the socket is currently connected to a remote host.

Parameters

None.

Return Value

If the client is connected to a remote host, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

CSocketWrench, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsListening Method

```
BOOL IsListening();
```

The **IsListening** method determines if the socket is listening for connection requests.

Parameters

None.

Return Value

If the socket is being used to listen for connection requests, the method returns a non-zero value. If the socket is not listening or the socket handle is invalid, the method returns zero.

Remarks

The **IsListening** method determines if the socket is being used in a server application to actively listen for incoming connection requests from client applications. A listening socket can be created using the **Listen** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, IsWritable, IsConnected, Listen

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsProtocolAvailable Method

```
BOOL IsProtocolAvailable(
   INT nAddressFamily,
   INT nProtocol
);
```

The **IsProtocolAvailable** method determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the system is capable of creating a socket using the specified address family and protocol, this method will return a non-zero value. If the combination of address family and protocol is not supported, this method will return a value of zero.

Remarks

The **IsProtocolAvailable** method is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!pSocket->IsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetAddress, GetHostAddress, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the current thread can read data from the socket without blocking, the method returns a non-zero value. If the current thread cannot read any data without blocking, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in ***lpdwAvail*** will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsWritable, Peek, Read, ReadLine, ReadStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsUrgent Method

```
BOOL IsUrgent();
```

The **IsUrgent** method determines if there is any out-of-band (OOB) data available to be read.

Parameters

None.

Return Value

If there is out-of-band data, the return value is non-zero. If there is no out-of-band data, or an error occurs the return value is zero. To determine if an error has occurred, use the **GetLastError** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

SetOption (INET_OPTION_INLINE)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If data can be written to the socket within the specified timeout period, the method returns a non-zero value. The method will return zero if the socket send buffer is full.

Remarks

The **IsWritable** method cannot be used to determine the amount of data that can be sent to the remote host without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **WriteStream** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

IsReadable, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Listen Method

```
SOCKET Listen(
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort,
   UINT nBacklog,
   DWORD dwOptions
);
```

The **Listen** method creates a listening socket and specifies the maximum number of connection requests that will be queued.

This method has been deprecated and is included for backwards compatibility. Use the **CInternetServer** class to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of queue of pending connections. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are recognized:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to

	the address provided by the caller, this function will fail.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.

Return Value

If the method succeeds, the return value is a handle to the socket. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

If the INET_OPTION_EXCLUSIVE option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the INET_OPTION_REUSEADDRESS option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Accept, DisableEvents, EnableEvents, Reject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::MatchHostName Method

```
BOOL MatchHostName(
   LPCTSTR lpszHostName,
   LPCTSTR lpszHostMask
   BOOL bResolve
);
```

The **MatchHostName** method matches a host name against one more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAddress, GetHostAddress, GetHostName, GetLocalAddress, GetPeerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NormalizeHostName Method

```
INT NormalizeHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszNormalized,
   INT nMaxLength
);

NormalizeHostName(
   LPCTSTR lpszHostName,
   CString& strNormalized
);
```

The **NormalizeHostName** method returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name. An alternate form of this method will accept a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

The maximum number of characters that can be copied to the *lpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **NormalizeHostName** method will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this method, it will be converted to an ASCII compatible format for domain names.

The *lpszHostName* parameter should only specify a host name or IP address. If you want to support the use of URLs to establish a connection, use the **GetUrlHostName** method which has extended support for extracting the host name and port number specified in a URL.

If the Unicode version of this method is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the **ValidateHostName** method.

It is recommended that you use this method if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this method prior to calling the other SocketWrench methods which accept a host name as a parameter. Those methods already normalize the host name and perform checks to ensure it is in the correct format.

If the *lpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the **GetHostName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHostAddress, GetHostName, GetUrlHostName, HostNameToUnicode, ValidateHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Peek Method

```
INT Peek(
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **Peek** method reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be stored. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the ***lpBuffer*** parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, ReadLine, RegisterEvent, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer,
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnRemotePort
);
```

The **Read** method reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be stored.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

lpAddress

Pointer to an INTERNET_ADDRESS structure that which will contain the IP address of the remote host that sent the data being read. If this information is not required, the parameter may be specified as NULL.

lpnRemotePort

Pointer to an unsigned integer which will contain the remote port number. If this information is not required, the parameter may be specified as NULL.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Read** method will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this method is invoked, the session thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This method will return if any amount of data is sent by the remote host, and will not block until the entire buffer has been filled. To avoid blocking the current thread, either create an asynchronous socket or use the **IsReadable** method to determine if there is data available to be read prior to calling this function.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the remote host is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** method will return only a partial block of data, or it

may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the remote host is sending string data to the server, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the remote host is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **ReadLine** method will return a line of text at a time and perform this conversion for you.

When **Read** is called and the socket is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Peek, ReadLine, RegisterEvent, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::ReadLine Method

```
BOOL ReadLine(
   LPTSTR lpszBuffer,
   LPINT lpnLength
);

BOOL ReadLine(
   LPTSTR lpszBuffer,
   INT nMaxLength
);

BOOL ReadLine(
   CString& strBuffer,
   INT nMaxLength
);
```

The **ReadLine** method reads up to a line of data from the socket and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the socket and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If this method is

called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pSocket->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pSocket->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsReadable, Peek, Read, ReadStream, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::ReadStream Method

```
BOOL ReadStream(
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions,
   LPBYTE lpMarker,
   DWORD cbMarker
);
```

The **ReadStream** method reads the socket data stream and stores the contents in the specified buffer.

Parameters

lpvBuffer

Pointer to the buffer that will contain or reference the data when the method returns. The actual argument depends on the value of the ***dwOptions*** parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the method returns. This argument should should always point to an initialized value. If the lpvBuffer argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the ***lpvBuffer*** parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the method will return a handle which references the data; otherwise, the method will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the method returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The ***lpvBuffer*** argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the method returns. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The ***lpvBuffer*** argument specifies a pointer to a global memory handle. When the method returns, the handle will reference a block of memory that contains the stream data. The application

	should take care to make sure that the handle passed to the method does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this method.
INET_STREAM_HANDLE	The ***lpvBuffer*** argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The ***lpvBuffer*** argument specifies a socket handle. The data read from the socket specified by the ***hSocket*** argument will be written to this socket. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this method.

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the method will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the ***lpMarker*** parameter is NULL, then this value must be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call **GetLastError**.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **Read** method, which will return immediately when any amount of data has been read, **ReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
HGLOBAL hgblBuffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400;   // Read up to 100K bytes
BOOL bResult;

bResult = pSocket->ReadStream(&hgblBuffer, &cbBuffer,
                              INET_STREAM_HGLOBAL | INET_STREAM_CONVERT);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Use data in the stream buffer

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetStreamInfo, Read, ReadLine, StoreStream, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   INETEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed.
INET_EVENT_DISCONNECT	The remote host has closed the connection to the client. The client should read any remaining data and disconnect.
INET_EVENT_READ	A network event which indicates data is available to read. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	A network event which indicates the application can send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR.

To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **InetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, InetEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Reject Method

```
BOOL Reject();
```

The **Reject** method is used to reject a client connection request.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Accept, Listen

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::SetHostFile Method

```
INT SetHostFile(
   LPCTSTR lpszFileName
);
```

The **SetHostFile** method specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the method succeeds, the return value is the number of entries in the host file. A return value of INET_ERROR indicates failure. To get extended error information, call **GetLastError**.

Remarks

This method loads the file into memory allocated for the current thread. If the contents of the file have changed after the method has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this method again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **GetHostFile** method. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **GetDefaultHostFile** method. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDefaultHostFile, GetHostAddress, GetHostFile, GetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::SetOption Method

```
INT SetOption(
   DWORD dwOption,
   BOOL bEnabled
);
```

The **SetOption** method is used to enable or disable a specific socket option.

Parameters

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

bEnabled

A boolean flag. If the flag is set to a non-zero value, the option is enabled. Otherwise the socket option is disabled.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

If if the INET_OPTION_KEEPALIVE option is enabled, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Connect, GetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Shutdown Method

```
INT Shutdown(
   DWORD dwOption
);
```

The **Shutdown** method is used to disable reception or transmission of data, or both.

Parameters

dwOption

An unsigned integer used to specify one of the shutdown options. These options cannot be combined. The following values are recognized:

Value	Description
INET_SHUTDOWN_READ	Disable reception of data.
INET_SHUTDOWN_WRITE	Disable transmission of data.
INET_SHUTDOWN_BOTH	Disable both reception and transmission of data.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method is rarely needed. It is provided as an interface to the Windows Sockets **shutdown** method.

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **Shutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

Abort, Connect, Disconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::StoreStream Method

```
BOOL StoreStream(
   LPCTSTR lpszFileName,
   DWORD dwLength,
   LPDWORD lpdwCopied
   DWORD dwOffset,
   DWORD dwOptions
);
```

The **StoreStream** method reads the socket data stream and stores the contents in the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the method will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the method returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Value	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the

	source data stream. If the *dwOffset* parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than memory buffers or handles.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
DWORD dwCopied;
BOOL bResult;

bResult = pSocket->StoreStream(lpszFileName, 0, &dwCopied, 0,
                               INET_STREAM_CONVERT);

if (bResult && dwCopied > 0)
{
    // The data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetStreamInfo, Read, ReadLine, ReadStream, Write, WriteLine, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::ValidateCertificate Method

```
BOOL ValidateCertificate(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertPassword,
   LPCTSTR lpszCertName
);
```

The **ValidateCertificate** method determines if the specified security certificate is installed on the local system.

Parameters

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the personal certificate store will be used as the default. This parameter may also specify the name of a certificate file in PKCS #12 (PFX) format.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertPassword

A null-terminated string which specifies the password associated with a certificate file. This parameter is only used if the ***lpszCertStore*** parameter specifies a certificate file, otherwise it is ignored. If the certificate file is not protected with a password, this parameter should be a NULL pointer or empty string.

lpszCertName

A null-terminated string which specifies the name of the certificate to validate. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

If you are checking the validity of a certificate installed in the local certificate store, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user.

It is possible to validate a certificate file rather than one stored in the local certificate store. The *lpszCertStore* member should specify the name of a file in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate file was created, it must be provided in with the *lpszCertPassword* parameter or this method will be unable to access the certificate.

This method can only validate certificate files in PFX format and cannot be used to validate a certificate file in another format, such as PEM (base64 encoded) or DER (binary).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials, DeleteSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ValidateHostName Method

```
BOOL ValidateHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszAddress,
   INT nMaxLength
);

BOOL ValidateHostName(
   LPCTSTR lpszHostName,
   CString& strAddress
);
```

The **ValidateHostName** method determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The method will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the ***nMaxLength*** parameter is zero, it will be ignored and the IP address will not be returned. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the ***lpszAddress*** string buffer. The buffer must be large enough to store the complete address. Because this method can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the ***lpszAddress*** parameter will be ignored.

Return Value

If the method succeeds, the host name is valid and the return value will be non-zero. If the method fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ValidateHostName** method provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. It is similar to calling the **NormalizeHostName** method to obtain the canonical form of the host name, calling **GetAddress** to obtain the IP address and then calling **FormatAddress** to return the string representation of the host's IP address.

If the Unicode version of this method is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

The ***lpszHostName*** parameter can only specify a host name or IP address and cannot be a URL. If you want your application to support providing a URL in addition to a host name, use the **GetUrlHostName** method to extract the host name from the URL. You can then provide the host

name to this method to obtain its IP address.

If the *lpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the **GetHostName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FormatAddress, GetAddress, GetHostAddress, GetHostName, GetUrlHostName, NormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer,
   LPINTERNET_ADDRESS lpAddress,
   UINT nRemotePort
);
```

The **Write** method sends the specified number of bytes to the remote host.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero.

lpAddress

Pointer to an INTERNET_ADDRESS structure that specifies the address of the remote host that is to receive the data being written. For TCP stream sockets, this parameter must always be NULL or specify the same address that was used to establish the connection. For UDP datagram sockets, this may specify any valid IP address.

nRemotePort

The port number of the remote host that is to receive the data being written. For TCP stream sockets, this value must always be zero, or specify the same port number that was used to establish the connection. For UDP datagram sockets, this may specify any valid port number.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking sockets, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, ReadLine, RegisterEvent, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::WriteLine Method

```
BOOL WriteLine(
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **WriteLine** function sends a line of text to the remote host, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a null-terminated string.

If the ***lpszBuffer*** string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the ***lpnLength*** parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application

needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsWritable, Read, ReadLine, ReadStream, Write, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CSocketWrench::WriteStream Method

```
BOOL WriteStream(
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);
```

The **WriteStream** method writes data from the stream buffer to the specified socket.

Parameters

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the ***dwOptions*** parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the method returns. This argument should should always point to an initialized value. If the ***lpvBuffer*** argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the ***lpvBuffer*** parameter. If the argument specifies a a global memory handle, then the method will write the data referenced by that handle; otherwise, the method will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The ***lpvBuffer*** argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The ***lpvBuffer*** argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The ***lpvBuffer*** argument specifies a Windows handle to

	an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The ***lpvBuffer*** argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the ***hSocket*** parameter. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **Write** method, which may not write all of the data in a single method call, **WriteStream** will only return when all of the data has been written or an error occurs.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **WriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for some data to have been written even if the method returns a value of zero. Applications should also check the value of the ***lpdwLength*** argument to determine if any data was sent. For example, if a timeout occurs while the method is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **WriteStream** can potentially cause the application to block for long periods of time as the data stream is being written, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
CFile *pFile = new CFile();
DWORD dwLength = 0;

if (!pFile->Open(strFileName, CFile::modeRead | CFile::shareDenyWrite))
    return;
```

```
dwLength = pFile->GetLength();

if (dwLength > 0)
{
    BOOL bResult = pSocket->WriteStream(
                      pFile->m_hFile,
                      &dwLength,
                      INET_STREAM_HANDLE);
}

delete pFile;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

GetStreamInfo, Read, ReadLine, ReadStream, StoreStream, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS
- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO

The INETSTREAMINFO structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD   dwStreamThread;
    DWORD   dwStreamSize;
    DWORD   dwStreamCopied;
    DWORD   dwStreamMode;
    DWORD   dwStreamError;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Value	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

See Also

ReadStream, StoreStream, WriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Class Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of method parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to method
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to method
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to method
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to method
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified

0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence

0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This method has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session

0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol Class Library

Establish an interactive terminal session with a server.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CTelnetClient
File Name	CSTNTV12.DLL
Version	12.0.3024.1286
LibID	552A583F-46C0-4A66-8BC5-B75B468763EE
Import Library	CSTNTV12.LIB
Dependencies	None
Standards	RFC 854

Overview

The Telnet protocol is used to establish a connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The class provides an interface for establishing the connection, negotiating certain options (such as whether characters will be echoed back to the client) and handling the standard I/O functions needed by the program.

The class also provides methods that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. The CTelnetClient class can be combined with the CTerminalEmulator class to provide complete terminal emulation services for a standard ANSI or DEC-VT220 terminal.

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the CSshClient class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol Class Methods

Class	Description
CTelnetClient	Constructor which initializes the current instance of the class
~CTelnetClient	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the current session and close the connection with the server
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Break	Send a break signal to the server
Cancel	Cancel the current blocking operation
Connect	Connect to the specified server
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of network function calls to a file
FreezeEvents	Suspend asynchronous event processing
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetMode	Return the current client mode
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetTerminalType	Return the current terminal type
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsThere	Determine if the server is available
IsWritable	Determine if data can be written to the server

Login	Login to the server using the specified username and password
Read	Read data returned by the server
ReadLine	Read a line of text from the server and return it in a string buffer
RegisterEvent	Register an event callback function
Search	Search for a specific character sequence in the data stream
SetLastError	Set the last error code
SetMode	Set the current client mode
SetTerminalType	Set the current terminal type
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
TelnetEventProc	Callback function that processes events generated by the client
Write	Write data to the server
WriteLine	Write a line of text to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::CTelnetClient Method

```
CTelnetClient();
```

The **CTelnetClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **TelnetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CTelnetClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::~CTelnetClient

```
~CTelnetClient();
```

The **CTelnetClient** destructor releases resources allocated by the current instance of the **CTelnetClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CTelnetClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CTelnetClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Abort Method

```
INT Abort();
```

The **Abort** method aborts the current session and terminates the connection.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Abort** method is called, the abort sequence is sent to the server and the connection to the server is terminated. Once this method returns, the client handle is no longer valid. If a program is currently executing on the server at the time this method is called, that program may be terminated as a result of the session being aborted. Applications should normally call **Disconnect** to gracefully disconnect from the server and should only use this method when the connection must be aborted immediately.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

Break, Cancel, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Break Method

```
INT Break();
```

The **Break** method sends a signal to the server which may terminate an application that is currently running. The actual response to the break signal depends on the application.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Abort, Cancel, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

Abort, Break, IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 23. For secure connections, the default port number is 992.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TELNET_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
TELNET_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
TELNET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
TELNET_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
TELNET_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a

	standard connection with the server, then requests a secure connection during the option negotiation process.
TELNET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
TELNET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
TELNET_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the CSshClient class.

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a

worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the TELNET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials, Disconnect, GetSecurityInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertName
);

BOOL CreateSecurityCredentials(
   LPCTSTR lpszCertName
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** parameter should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pClient->CreateSecurityCredentials(
            SECURITY_PROTOCOL_DEFAULT,
            0,
            NULL,
            NULL,
            lpszCertStore,
            lpszCertName);

bConnected = pClient->Connect(lpszHostName,
                              TELNET_PORT_SECURE,
                              TELNET_TIMEOUT,
                              TELNET_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, DeleteSecurityCredentials, GetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetMode Method

```
UINT GetMode();
```

The **GetMode** method returns the current client mode.

Parameters

None.

Return Value

If the method succeeds, the return value is the current client mode. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client mode is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters and single linefeeds will be automatically converted to carriage-return/linefeed sequences.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(
   LPSECURITYINFO lpSecurityInfo
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;
securityInfo.dwSize = sizeof(SECURITYINFO);

if (pClient->GetSecurityInformation(&securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, CreateSecurityCredentials, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
TELNET_STATUS_UNUSED	No connection has been established.
TELNET_STATUS_IDLE	The client is current idle and not sending or receiving data.
TELNET_STATUS_CONNECT	The client is establishing a connection with the server.
TELNET_STATUS_READ	The client is reading data from the server.
TELNET_STATUS_WRITE	The client is writing data to the server.
TELNET_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

IsBlocking, IsConnected, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetTerminalType Method

```
INT GetTerminalType(
   LPCTSTR lpszTermType,
   INT nMaxLength
);

INT GetTerminalType(
   CString& strTermType
);
```

The **GetTerminalType** method returns the terminal type for the current client session.

Parameters

lpszTermType

Points to a buffer which the current terminal type is copied into. This buffer should be at least 32 characters in length, including the terminating null character. This argument may also be a **CString** object which will contain the terminal type when the method returns.

nMaxLength

Maximum number of characters that may be copied to the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the terminal type name. A value of zero indicates that no terminal type has been specified. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SetTerminalType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

Connect, IsReadable, IsWritable, Read, SetTimeout, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

CTelnetClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsWritable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::IsThere Method

```
BOOL IsThere();
```

The **IsThere** method reports the response of the client to a "Are you there" command to the telnet server.

Parameters

None.

Return Value

The method returns a non-zero value if the server acknowledges a specific control sequence used to determine if a server is responsive. If the server does not respond, the method will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsReadable, IsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::IsWritable Method

```
BOOL IsWritable(
   INT nTimeout
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized, IsReadable, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Login Method

```
BOOL Login(
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **Login** method attempts to authenticate the user and log them in to the current session.

Parameters

lpszUserName

A null-terminated string which specifies the name of the user to authenticate.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the user. If the user does not require a password, this parameter may be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the CSshClient class.

This method is used to authenticate a user, logging them into the server. This method is specifically designed to work with most UNIX based servers, and may work with other servers that use a similar login process. The method works by scanning the data stream for a username prompt and then replying with the specified username. If that is successful, it will then scan for a password prompt and provide the specified password. If no recognized prompt is found, or if the server responds with an error indicating that the username or password is invalid, the method will fail.

If the **Login** method succeeds, the next call to **Read** by the client will return any welcome message to the user. This is typically followed by a command prompt where the user can enter commands to be executed on the server. The data sent by the server during the login process is discarded and not available when the method returns. If the client requires this information, use the **Search** method to automate the login process instead.

Because the **Login** method is designed for UNIX based systems, it may not work with servers running on other operating system platforms such as Windows or VMS. In this case, applications should use the **Search** method to search for the appropriate login prompts in the data stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, IsConnected, IsReadable, Read, Search

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, RegisterEvent, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::ReadLine Method

```
BOOL ReadLine(
   LPTSTR lpszBuffer,
   LPINT lpnLength
);

BOOL ReadLine(
   LPTSTR lpszBuffer,
   INT nMaxLength
);

BOOL ReadLine(
   CString& strBuffer,
   INT nMaxLength
);
```

The **ReadLine** method reads up to a line of data from the server and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the server and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the socket connection. If this method is called

with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pClient->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pClient->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsReadable, Read, Write, WriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   TELNETEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
TELNET_EVENT_CONNECT	The connection to the server has completed.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **TelnetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **TelnetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, TelnetEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Search Method

```
BOOL Search(
   LPCTSTR lpszString
);

BOOL Search(
   LPCTSTR lpszString,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength
);

BOOL Search(
   LPCTSTR lpszString,
   HGLOBAL *lpBuffer,
   LPDWORD lpdwLength
);

BOOL Search(
   LPCTSTR lpszString,
   CString& strBuffer
);
```

The **Search** method searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

lpszString

A null-terminated string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpBuffer

A buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the method returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpBuffer*** parameter. If the ***lpBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the ***lpBuffer*** parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds and the character sequences was found in the data stream, the return value is non-zero. If the method fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as logging in a user and executing a command. The method collects the output from the server and stores it in the buffer specified by the ***lpBuffer*** parameter. When the method returns, the buffer will contain everything sent by the server up to and including the search string.

The *lpBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the output when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
CString strOutput;
BOOL bResult;

// Search for the login prompt issued by the server

bResult = pClient->Search(T("ogin: ");

// If the Login: prompt was found, then write out the
// username and search for the Password: prompt; note
// that the username, password and command strings are
// terminated with a carriage-return/linefeed sequence
// which the server will see as the user pressing the
// Enter or Return key on the keyboard

if (bResult)
{
    pClient->Write(lpszUserName);
    bResult = pClient->Search(_T("word: "));
}

// If the Password: prompt was found, write out the
// password and then search for the shell prompt;
// the prompt may be different, depending on what
// operating system and shell is being used

if (bResult)
{
    pClient->Write(lpszPassword);
    bResult = Search(hClient, _T("$ "));
}

// If the shell prompt was found, issue the command
// and capture the output into the strOutput string
// and then write it to standard output

if (bResult)
{
    pClient->Write(lpszCommand);

    if (pClient->Search(_T("$ "), strOutput))
        cout << (LPCTSTR)strOutput << endl;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsBlocking, IsReadable, Login, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::SetMode Method

```
UINT SetMode(
   UINT nMode,
   BOOL bEnable
);
```

The **SetMode** method sets one or more client modes for the specified session.

Parameters

nMode

The client mode. This value is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters, and single linefeed characters will be converted to carriage-return/linefeed sequences.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the method succeeds, the return value is the previous mode. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

GetMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::SetTerminalType Method

```
INT SetTerminalType(
   LPCTSTR lpszTermType
);
```

The **SetTerminalType** method sets the terminal type for the current client session.

Parameters

lpszTermType

Points to a string which specifies the terminal type.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetTerminalType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

Connect, GetTimeout, IsReadable, IsWritable, Read, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::Write Method

```
INT Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Write(
   LPCTSTR lpszBuffer
   INT cbBuffer
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableEvents, IsBlocking, IsReadable, IsWritable, Read, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTelnetClient::WriteLine Method

```
BOOL WriteLine(
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **WriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a null-terminated string.

If the ***lpszBuffer*** string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the ***lpnLength*** parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to

establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsWritable, Read, ReadLine, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation Class Library

Emulate an ANSI or DEC VT-220 character mode display terminal.

Reference

- Class Methods
- Data Structures
- Control Sequences
- Error Codes

Library Information

Class Name	CTerminalEmulator
File Name	CSNVTV12.DLL
Version	12.0.3024.1286
LibID	9A302CB9-6BEE-4D11-9785-76E7624BD130
Import Library	CSNVTV12.LIB
Dependencies	None

Remarks

The Terminal Emulation library provides a comprehensive API for emulating an ANSI or DEC-VT220 terminal, with full support for all standard escape and control sequences, color mapping and other advanced features. The library methods provide both a high level interface for parsing escape sequences and updating a display, as well as lower level primitives for directly managing the virtual display, such as controlling the individual display cells, moving the cursor position and specifying display attributes.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing

applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation Class Methods

Class	Description
CTerminalEmulator	Constructor which initializes the current instance of the class
~CTerminalEmulator	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified display handle to this instance of the class
Clear	Clear the virtual display
ConvertPos	Convert between pixel and screen coordinates
CopySelectedText	Copy the selected text to the clipboard
Create	Create a new virtual terminal emulation display
DeleteChar	Delete the specified number of characters
DeleteLine	Delete the current line, shifting remaining lines up
DetachHandle	Detach the handle for the current instance of this class
EraseChar	Erase the specified number of characters
EraseLine	Erase the current line
GetAttributes	Return the current display attributes
GetBackColor	Return the current background color
GetCell	Return information about the specified cell in the virtual display
GetCellSize	Return the size of the display character cells
GetColor	Get the current virtual display colors
GetColorMap	Return the virtual display color table
GetCursorPos	Return the current cursor position
GetDisplayInfo	Return information about the virtual display
GetDC	Get the current virtual display device context
GetEmulation	Get the current terminal emulation type
GetFont	Get the current virtual display font
GetForeColor	Return the current foreground color
GetLine	Return a line of text from the virtual display
GetMode	Get the current virtual display mode
GetRect	Return the rectangle for the display window client area
GetScrollPos	Get the display scroll box position
GetSize	Return the current size of the virtual display
GetText	Get the specified block of text from the virtual display

GetWindow	Get the current virtual display window
GetHandle	Return the display handle used by this instance of the class
GetMappedKey	Return the escape sequence for the mapped key
GetScrollRegion	Return the current scrolling region
GetSelectedText	Return the currently selected text
GetTextColor	Return the current text foreground or background color
InsertChar	Insert the specified number of characters
InsertLine	Insert a line, shifting the remaining lines down
IsInitialized	Determine if the class has been successfully initialized
Refresh	Refresh the specified display
Reset	Reset the virtual display
ResetMappedKeys	Reset the mapped key table to default values
Resize	Resize the virtual display
RestoreCursor	Restore the saved cursor position and text attributes
SaveCursor	Save the current cursor position and text attributes
Scroll	Scroll the virtual display
SelectText	Select a region of the virtual display
SetAttributes	Set the current display attributes
SetBackColor	Set the background color for the virtual display
SetBoldColor	Set the bold color for the virtual display
SetCell	Set the value of a character cell in the virtual display
SetColor	Set the virtual display colors
SetColorMap	Set the virtual display color table
SetCursorPos	Set the current cursor position
SetDC	Set the current virtual display device context
SetEmulation	Set the current terminal emulation type
SetFocus	Set the focus on the virtual display
SetFont	Set the current virtual display font
SetForeColor	Set the foreground color for the virtual display
SetMode	Set the current virtual display mode
SetScrollPos	Set the display scroll box position
SetSize	Set the size of the virtual display
SetWindow	Set the current virtual display window
SetMappedKey	Set the escape sequence for the specified key
SetScrollRegion	Set the current scrolling region

SetTextColor	Set the current text foreground or background color
TranslateMappedKey	Translate the keypress to a mapped key escape sequence
Update	Update the window attached to the virtual display
UpdateCaret	Update the display window caret
Write	Write the specified buffer to the virtual display

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::CTerminalEmulator Method

```
CTerminalEmulator();
```

The **CTerminalEmulator** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **NvtInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CTerminalEmulator, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::~CTerminalEmulator

```
~CTerminalEmulator();
```

The **CTerminalEmulator** destructor releases resources allocated by the current instance of the **CTerminalEmulator** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CTerminalEmulator** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the emulator are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CTerminalEmulator

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::AttachHandle Method

```
VOID AttachHandle(
   HDISPLAY hDisplay
);
```

The **AttachHandle** method attaches the specified display handle to the current instance of the class.

Parameters

hDisplay

The handle to the virtual display that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a display handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that virtual display.

The virtual display for the current instance of the class will be destroyed when the new handle is attached to the class object. If you want to prevent the display handle from being released, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hDisplay*** parameter is presumed to be a valid display handle and no checks are performed to ensure that the handle is valid. Specifying an invalid handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Clear Method

```
BOOL Clear(
   UINT nMode
);
```

The **Clear** method clears the specified display, erasing the text and clearing any attributes.

Parameters

nMode

Mode which specifies how the display will be cleared. The following values may be used:

Value	Description
NVT_CLEAR_EOS	The display is cleared from the current cursor position to the end of the display. The cursor position is not changed.
NVT_CLEAR_TOS	The display is cleared from the beginning of the display to the current cursor position. The cursor position is not changed.
NVT_CLEAR_ALL	The entire display is cleared and the cursor is repositioned to the upper left corner of the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

DeleteChar, DeleteLine, EraseChar, EraseLine, Reset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::ConvertPos Method

```
BOOL ConvertPos(
   INT nMethod,
   INT xPos,
   INT yPos,
   LPPOINT lppt
);

BOOL ConvertPos(
   INT xPos,
   INT yPos,
   LPPOINT lppt
);
```

The **ConvertPos** method converts the specified X,Y position and stores the result in the POINT structure provided by the caller.

Parameters

nMethod

An integer value which specifies the conversion method to use. This may be one of the following values:

Value	Description
NVT_CURSOR_TO_PIXELS	Convert cursor X,Y coordinates to the current window X,Y pixel coordinates. An error is returned if the coordinates are out of bounds for the current display.
NVT_PIXELS_TO_CURSOR	Convert window X,Y pixel coordinates to cursor X,Y coordinates. If the point is outside of the bounds of the display, it is normalized to account for mouse capture.

xPos

An integer value which specifies the X position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the ***nMethod*** parameter.

yPos

An integer value which specifies the Y position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the ***nMethod*** parameter.

lppt

A pointer to a POINT structure which will contain the converted coordinates.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetCursorPos, GetCellSize, GetScrollPos, GetSize, SetCursorPos, SetScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::CopySelectedText Method

```
BOOL CopySelectedText();
```

The **CopySelectedText** method copies any selected text to the system clipboard.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetSelectedText, SelectText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Create Method

```
BOOL Create(
   HWND hWnd,
   HFONT hFont,
   UINT nEmulation,
   UINT nColumns,
   UINT nRows
);
```

The **Create** method creates a new virtual terminal emulation display using the specified window and font.

Parameters

hWnd

Handle to the window that will be used to display the virtual terminal.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process escape sequences.
NVT_EMULATION_ANSI	The virtual display processes ANSI escape sequences for screen management and cursor positioning. This emulation also supports escape sequences to control the foreground and background color. The default keymap for ANSI function key escape sequences will be selected.
NVT_EMULATION_VT100	The virtual display processes DEC VT-100 escape sequences for screen management and cursor positioning. The default keymap for a DEC VT-100 terminal will be selected.
NVT_EMULATION_VT220	The virtual display processes DEC VT-220 escape sequences for screen management and cursor positioning. This emulation also supports DEC VT-320 escape sequences to control the foreground and background color. The default keymap for a DEC VT-220 terminal will be selected.

nColumns

The maximum number of columns used by the virtual display. This value must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. This value must be at least 5, and no greater than 127.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The default colors for the display is a black background and white foreground. If ANSI terminal emulation is selected, bold characters will be displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

AttachHandle, DetachHandle, GetDisplayInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::DeleteChar Method

```
BOOL DeleteChar(
   INT nChars
);
```

The **DeleteChar** method deletes the specified number of characters from the display, shifting the characters that follow to the left. The characters are deleted from the current cursor position.

Parameters

nChars

Number of characters to delete from the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To erase characters from the display without affecting the characters that follow, use the **EraseChar** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Clear, DeleteLine, EraseChar, EraseLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::DeleteLine Method

```
BOOL DeleteLine();
```

The **DeleteLine** method deletes the current line, shifting up the lines that follow.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To erase a line from the display without affecting the lines that follow, use the **EraseLine** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Clear, DeleteChar, EraseChar, EraseLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::DetachHandle Method

```
HDISPLAY DetachHandle();
```

The **DetachHandle** method detaches the display handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the display handle associated with the current instance of the class object. If there is no virtual display associated with that instance of the class, the value INVALID_DISPLAY will be returned.

Remarks

This method is used to detach a display handle created by the class for use with the SocketTools API. Once the handle is detached from the class, no other class methods may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::EraseChar Method

```
BOOL EraseChar(
   INT nChars
);
```

The **EraseChar** method erases the specified number of characters from the display, without affecting the position of the characters that follow. The characters are erased from the current cursor position.

Parameters

nChars

Number of characters to erase from the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To delete characters from the display and shift the remaining characters to the left, use the **DeleteChar** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Clear, DeleteChar, DeleteLine, EraseLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::EraseLine Method

```
BOOL EraseLine(
   UINT nMode
);
```

The **EraseLine** method erases the current line without affecting the lines that follow.

Parameters

nMode

Mode which specifies how the line will be erased. The following values may be used:

Value	Description
The line is erased from the current cursor position to the end of the line.	
The line is cleared from the beginning of the line to the current cursor position.	
The entire line is cleared.	

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To delete a line from the display and shift the remaining lines up, use the **DeleteLine** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Clear, DeleteChar, DeleteLine, EraseChar

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetAttributes Method

```
UINT GetAttributes();
```

The **GetAttributes** method returns the current display attributes which have been set, either explicitly by the client or as the result of an escape sequence parsed by the emulator.

Parameters

None.

Return Value

If the method succeeds, the return value the current display attributes. If the method fails, the return value is NVT_ERROR.

The following table lists the valid attributes:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetCell, SetAttributes

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetBackColor Method

```
COLORREF GetBackColor();
```

The **GetColor** method returns the background color used by the current display.

Parameters

None.

Return Value

If the method succeeds, the return value is the RGB value of the background color. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetForeColor, SetBackColor, SetBoldColor, SetColor, SetBackColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetCell Method

```
BOOL GetCell(
   INT xPos,
   INT yPos
   TCHAR *pChar
   UINT *pAttributes
);

BOOL GetCell(
   TCHAR *pChar
   UINT *pAttributes
);
```

The **GetCell** method returns information about the specified character cell in the virtual display.

Parameters

xPos

An integer value which specifies the X cursor position in the display. If this argument is omitted or has the value -1, then the current position in the display is used.

yPos

An integer value which specifies the Y cursor position in the display. If this argument is omitted or has the value -1, then the current position in the display is used.

pChar

A buffer that will contain the character in the specified cell when the method returns. If this argument is NULL, then no character is returned.

pAttributes

A pointer to an unsigned integer value which specifies the attributes for the specified character cell. If this argument is NULL, then no attribute information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero. Failure typically indicates that the X or Y position is invalid, or that a virtual display has not been created.

Remarks

The attribute value returned by the **GetCell** method may be one or more of the following values:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as

NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetAttributes, GetCellSize, SetAttributes

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetCellSize Method

```
BOOL GetCellSize(
   LPSIZE lpCellSize
);
```

```
BOOL GetCellSize(
   LPINT lpnWidth
   LPINT lpnHeight
);
```

The **GetCellSize** method returns the size of a character cell in pixels.

Parameters

lpCellSize

A pointer to a SIZE structure which will contain the size of a character cell in pixels when the method returns.

lpnWidth

A pointer to an integer which will contain the width of a character cell in pixels when the method returns.

lpnHeight

A pointer to an integer which will contain the height of a character cell in pixels when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero. Failure typically indicates that the virtual display has not been created.

Remarks

The **GetCellSize** method is used to determine the size of a character cell in the display. This can be useful when the application needs to determine where a display cell is physically located within the virtual display window.

To convert between display and window coordinates, use the **ConvertPos** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

ConvertPos, GetCell

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetColor Method

```
COLORREF GetColor(
   UINT nColorIndex,
   BOOL bForeground
);
```

The **GetColor** method returns the color used by the virtual display to display text with a specific attribute.

Parameters

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Value	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the method succeeds, the return value is the RGB value of the specified color. If the method fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetBackColor, GetForeColor, SetBackColor, SetBoldColor, SetColor, SetForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetColorMap Method

```
BOOL GetColorMap(
   COLORREF *lpColor,
   INT nMaxColors
);
```

The **GetColorMap** method returns the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

lpColor

A pointer to an array of COLORREF values which will contain the color values currently being used in the virtual display.

nMaxColors

The maximum number of colors which may be stored in the color array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid, the pointer to the color table is NULL or the maximum number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **GetColorMap** method is useful for determining what values are being used when a color attribute is set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
Black	RGB(0,0,0)			
Red	000000A0h	RGB(160,0,0)		
Green	0000A000h	RGB(0,160,0)		
Yellow	0000A0A0h	RGB(160,160,0)		
Blue	00A00000h	RGB(0,0,160)		
Magenta	00A000A0h	RGB(160,0,160)		
Cyan	00A0A000h	RGB(0,160,160)		
White	00E0E0E0h	RGB(224,224,224)		
Gray	00C0C0C0h	RGB(192,192,192)		
Light Red	008080FFh	RGB(255,128,128)		
Light Green	0090EE90h	RGB(144,238,144)		
Light Yellow	00C0FFFFh	RGB(255,255,192)		
Light Blue	00E6D8ADh	RGB(173,216,230)		

Light Magenta	00FFC0FFh	RGB(255,192,255)
Light Cyan	00FFFFE0h	RGB(224,255,255)
High White	00FFFFFFh	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

GetColorMap is typically used in conjunction with the **SetColorMap** method to load the current color values and then make selective changes to the actual RGB color value that is used when a color attribute is set. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if (pDisplay->GetColorMap(rgbColor, 16) )
{
     rgbColor[7] = rgbColor[15];
     pDisplay->SetColorMap(rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetColor, GetTextColor, SetColor, SetColorMap, SetTextColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetCursorPos Method

```
BOOL GetCursorPos(
   LPINT lpnCursorX,
   LPINT lpnCursorY
);
```

The **GetCursorPos** method returns the current cursor column and row position on the virtual display.

Parameters

lpnCursorX

Address of the variable that will be set to the column of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

lpnCursorY

Address of the variable that will be set to the row of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

Return Value

If the method succeeds, the return value is a non-zero value. If the handle to the virtual display is invalid, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

ConvertPos, SetCursorPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetDC Method

```
HDC GetDC();
```

The **GetDC** returns the device context that has been specified for the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is a handle to the device context. If the method fails, or no device context has been specified, the return value is NULL.

Remarks

Normally there is no device context explicitly set for the display. Instead, the device context is dynamically created and released when the virtual display is updated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

SetDC, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetDisplayInfo Method

```
BOOL GetDisplayInfo(
   LPNVTDISPLAYINFO lpvdi
);
```

The **GetDisplayInfo** method returns information about the specified virtual display.

Parameters

lpvdi

Pointer to a NVTDISPLAYINFO structure which contains information about the virtual display, including the display window, font, cursor and scrolling position.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Create, GetCursorPos, GetFont, GetMode, GetRect, GetScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetEmulation Method

```
UINT GetEmulation();
```

The **GetEmulation** method returns the current virtual terminal emulation type.

Parameters

None.

Return Value

If the method succeeds, the return value is the terminal emulation type and may contain one of the following values:

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

If the method fails, because an invalid display handle was specified, the return value will be NVT_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Create, Reset, SetEmulation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetFont Method

```
HFONT GetFont();
```

The **GetFont** method returns the current font handle being used by the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is a handle to the font. If the method fails, it returns NULL.

Remarks

The handle returned by this method should not be released, and the font object should never be directly modified by the application. Functions that return information about the font, such as **GetTextMetrics**, may be safely called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Create, SetFont

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetForeColor Method

```
COLORREF GetForeColor();
```

The **GetColor** method returns the foreground color used by the current display.

Parameters

None.

Return Value

If the method succeeds, the return value is the RGB value of the foreground color. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetBackColor, SetBackColor, SetBoldColor, SetColor, SetForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetHandle Method

```
HDISPLAY GetHandle();
```

The **GetHandle** method returns the display handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the display handle associated with the current instance of the class object. If there is no virtual display associated with that instance of the class, the value INVALID_DISPLAY will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetLine Method

```
INT GetLine(
   INT nRow,
   LPTSTR lpszText,
   INT nMaxLength
);

INT GetLine(
   INT nRow,
   CString& strText
);
```

The **GetLine** method copies a block of text from the specified virtual display into a string buffer.

Parameters

nRow

The row in the virtual display to return the line of text from. The first row in the display is zero. If the value -1 is specified, the row where the cursor is currently located will be used.

lpszText

Pointer to the buffer that the display text will be copied to, terminated with a null character character. An alternate version of this method accepts a reference to a **CString** object which will contain the display text when the method returns.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the method fails, it returns zero.

Remarks

The **GetLine** method allows the application to copy the contents of the display at a specific row. Note that the buffer must be large enough to accommodate the text and the null character terminator. Unlike the **GetText** method, any trailing whitespace in the specified row is not copied to the buffer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetText, Update, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetMappedKey Method

```
BOOL GetMappedKey(
   UINT nMappedKey,
   LPTSTR lpszKeyBuffer,
   UINT cchKeyBuffer
);
```

```
BOOL GetMappedKey(
   UINT nMappedKey,
   CString& strKeyBuffer
);
```

The **GetMappedKey** method returns the escape sequence mapped to the specified key.

Parameters

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (method) keys for the current emulation.

lpszKeyBuffer

Address of the buffer which will receive the escape sequence mapped to the specified key. This argument may also be a **CString** object which will contain the mapped key string when the method returns.

cchKeyBuffer

The maximum number of characters that may be copied into the key buffer string, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero and the escape sequence for the mapped key is copied into the specified buffer. If the key has not been mapped, and there is no default escape sequence defined, then the method will return zero.

Remarks

The **GetMappedKey** method returns the escape sequence that has been mapped to a special key. This method can be used to determine what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 method key). If a sequence has not been explicitly mapped through a call to **SetMappedKey**, the default sequence for the current emulation will be returned.

Note that the current display mode, such as whether or not the emulator is in application mode or not, should be considered when determining which mapped key to use. For example, if the emulator is not in application mode and the user presses the up-arrow key, the sequence mapped to the NVT_UP key should be sent to the server. However, if the emulator is in application mode, the sequence mapped to the NVT_APPUP key should be sent instead. This is automatically handled by the **TranslateMappedKey** method, so it is recommended that it be used when mapping a virtual keypress to the appropriate escape sequence.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMode, ResetMappedKeys, SetMode, SetMappedKey, TranslateMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetMode Method

```
UINT GetMode();
```

The **GetMode** method returns the current virtual display modes.

Parameters

None.

Return Value

If the method succeeds, the return value is the display mode for the virtual display. If the method fails, it returns zero.

Remarks

The display mode is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other methods. The following values are recognized.

Value	Description
NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT (horizontal tab) control

	sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less then that total number of columns in the virtual display. Disabling this mode prevents a horizontal scrollbar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less then that total number of rows in the virtual display. Disabling this mode prevents a vertical scrollbar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetEmulation, GetDisplayInfo, SetMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetRect Method

```
BOOL GetRect(
   LPRECT lpRect
);
```

The **GetRect** returns the client rectangle for the virtual display window.

Parameters

lpRect

Pointer to a RECT structure which will contain the client rectangle values when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetDisplayInfo, GetSize, Resize, SetSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetScrollPos Method

```
INT GetScrollPos(
   INT nScrollBar
);
```

The **GetScrollPos** method gets the position of the scroll box for the specified scroll bar.

Parameters

nScrollBar

Specifies the scroll bar to return the position for. This parameter can be one of the following values:

Value	Description
SB_HORZ	Gets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Gets the position of the scroll box in the display's standard vertical scroll bar.

Return Value

If the method succeeds, the return value is the position of the scroll box. If the method fails, it returns -1.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

SetScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetScrollRegion Method

```
BOOL GetScrollRegion(
   LPINT lpnTop,
   LPINT lpnBottom
);
```

The **GetScrollRegion** method returns the top and bottom rows of the current scrolling region.

Parameters

lpnTop

Address of the variable that will be set to the top row of the current scrolling region. If no scrolling region has been defined, the value will be 0, the first row in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

lpnBottom

Address of the variable that will be set to the bottom row of the current scrolling region. If no scrolling region has been defined, the value will be one less than the maximum number of rows in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

Return Value

If the method succeeds, it will return zero. If the handle to the display is invalid, the method will return zero.

Remarks

The **GetScrollRegion** method allows an application to determine the top and bottom rows of the current scrolling region. By default, the scrolling region is the entire virtual display, however this may be changed through a call to **SetScrollRegion** or an ANSI escape sequence. If the display is in origin mode, note that the cursor cannot be positioned outside of the current scrolling region.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetMode, SetMode, SetScrollRegion

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetSelectedText Method

```
INT GetSelectedText(
   LPTSTR lpszBuffer,
   INT nMaxLength
);

INT GetSelectedText(
   CString& strBuffer
);
```

The **GetSelectedText** method copies the currently selected text into the specified buffer.

Parameters

lpszBuffer

Pointer to the buffer that the selected text will be copied to, terminated with a null character character. This argument may also be a **CString** object which will contain the selected text when the method returns.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CopySelectedText, SelectText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetSize Method

```
BOOL GetSize(
   LPSIZE lpSize
);

BOOL GetSize(
   LPINT lpnWidth,
   LPINT lpnHeight
);
```

The **GetSize** returns the size of the virtual display in columns and rows.

Parameters

lpSize

Pointer to a SIZE structure which will contain the size of the virtual display.

lpnWidth

Pointer to an integer which will contain the number of columns in the virtual display when the method returns. If a NULL pointer is passed as the argument, this value is ignored.

lpnHeight

Pointer to an integer which will contain the number of rows in the virtual display when the method returns. If a NULL pointer is passed as the argument, this value is ignored.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

The **GetSize** method returns the number of columns and rows in the virtual display. To convert this to pixels, use the **GetCellSize** method to determine the size of a character cell and multiply that by the number of columns and/or rows.

Example

```
// Calculate the size of the client area in pixels
// for the virtual display

INT cxCell = 0, cyCell = 0;
INT cxDisplay = 0, cyDisplay = 0;

if (pDisplay->GetCellSize(&cxCell, &cyCell))
{
    INT nColumns, nRows;
    if (pDisplay->GetSize(&nColumns, &nRows))
    {
        cxDisplay = nColumns * cxCell;
        cyDisplay = nRows * cyCell;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csnvtv12.lib

See Also

GetCellSize, GetDisplayInfo, GetRect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetText Method

```
INT GetText(
   INT xPos,
   INT yPos,
   LPTSTR lpszText,
   INT nLength
);

INT GetText(
   INT xPos,
   INT yPos,
   CString& strText,
   INT nLength
);

INT GetText(
   LPTSTR lpszText,
   INT nLength
);

INT GetText(
   CString& strText,
   INT nLength
);
```

The **GetText** method copies a block of text from the specified virtual display into a string buffer.

Parameters

xPos

The starting column where the display text will be copied from. If the method is called where this argument is omitted, the text will be copied from the current cursor position.

yPos

The starting row where the display text will be copied from. If the method is called where this argument is omitted, the text will be copied from the current cursor position.

lpszText

Pointer to the buffer that the display text will be copied to, terminated with a null character character. An alternate version of this method accepts a reference to a **CString** object which will contain the display text when the method returns.

nLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator. If a CString object is passed to the method, this value may be -1 which specifies that all of the text from the specified position to the last column and row should be copied to the buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the method fails, it returns zero.

Remarks

The **GetText** method allows the application to copy the contents of the display at a specific location. Note that the buffer must be large enough to accommodate the text and the null character

terminator. To copy an entire row of text in the display, use the **GetLine** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLine, Update, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetTextColor Method

```
BOOL GetTextColor(
   COLORREF *lprgbColor,
   BOOL bForeground
);
```

The **GetTextColor** method returns the current foreground or background text color.

Parameters

lprgbColor

A pointer to a COLORREF variable which will contain the current foreground or background text color.

bForeground

A boolean value which determines if the foreground or background color is returned. A value of TRUE indicates that the foreground color should be returned, otherwise the background color is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. Failure indicates that the handle to the display is invalid, the display does not support text color attributes, or the pointer to the color value is NULL.

Remarks

This method is used to return the current foreground or background color, as determined by the text attribute. The RGB color value for a color attribute is determined by the virtual display's color table.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetColorMap, SetColorMap, SetTextColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetWindow Method

```
HWND GetWindow();
```

The **GetWindow** returns the handle to the window being used by the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is a handle to the virtual display window. If the method fails, it returns NULL.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetDisplayInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::InsertChar Method

```
BOOL InsertChar(
   INT nChars
);
```

The **InsertChar** method inserts one or more space characters at the current cursor position.

Parameters

nChars

Number of space characters to insert at the current cursor position.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the cursor position. Inserting space characters may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

DeleteChar, EraseChar, InsertLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::InsertLine Method

```
BOOL InsertLine();
```

The **InsertLine** method inserts an empty line at the current row, shifting the remaining lines down.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the cursor position. Inserting a line may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

DeleteLine, EraseLine, InsertChar

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

CTerminalEmulator

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Refresh Method

```
BOOL Refresh(
   BOOL bUpdate
);
```

The **Refresh** method refreshes the specified virtual display, updating the current scroll position and caret.

Parameters

bUpdate

Boolean flags which specifies if the display window is to be updated. If set, the window client area is invalidated and the virtual display is redrawn.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

UpdateCaret, Update, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Reset Method

```
BOOL Reset(
   HWND hWnd,
   HFONT hFont,
   UINT nColumns,
   UINT nRows
);

BOOL Reset(
   UINT nColumns,
   UINT nRows
);
```

The **Reset** resets the virtual terminal display, using the new window, font, columns and rows. This method should be used when the virtual display must be attached to a different window, or the number of rows or columns must be changed.

Parameters

hWnd

Handle to the window that will be used to display the virtual terminal. This parameter may be NULL, in which case the current window will be used.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case the current font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nColumns

The maximum number of columns used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 127.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

This method will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Clear, Create, Refresh, Resize, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::ResetMappedKeys Method

```
BOOL ResetMappedKeys();
```

The **ResetMappedKeys** method resets all mapped method keys to their default values, based on the current emulation.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the handle to the display is invalid, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetMappedKey, SetMappedKey, TranslateMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Resize Method

```
BOOL Resize(
   INT cxClient,
   INT cyClient
);
```

The **Resize** method resizes the virtual display to the specified width and height.

Parameters

cxClient

New width of the display window in pixels. If this value is zero, the width of the virtual display remains unchanged.

cyClient

New height of the display window in pixels. If this value is zero, the height of the virtual display remains unchanged.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method resizes the virtual display and updates the scrolling information. Typically this method is called when the display window receives a WM_SIZE message, causing the virtual display to match the size of the window's client area.

This method will not change the size of the display window. To change the size of the display window, use the **SetWindowPos** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Create, GetRect, Refresh, Reset, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::RestoreCursor Method

```
BOOL RestoreCursor();
```

The **RestoreCursor** method restores the cursor position and text attributes to their previous values.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

Use the **SaveCursor** method to save the current cursor position and text attributes. This method may only be called once for each time that the cursor position is saved.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetCursorPos, SaveCursor, SetCursorPos, UpdateCaret, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SaveCursor Method

```
BOOL SaveCursor();
```

The **SaveCursor** method saves the current cursor position and text attributes.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

Use the **RestoreCursor** method to restore the cursor position and text attributes to their previous values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetCursorPos, RestoreCursor, SetCursorPos, UpdateCaret, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Scroll Method

```
BOOL Scroll(
   BOOL bScrollUp
);
```

The **Scroll** method scrolls the virtual display up or down and updates the scroll box position if necessary.

Parameters

bScrollUp

Boolean flag which specifies if the virtual display is scrolled up or down.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. The **Refresh** method should be called to update the window attached to the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetScrollPos, Refresh, SetScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SelectText Method

```
BOOL SelectText(
   LPRECT lpRect,
   DWORD dwOptions
);
```

The **SelectText** method selects or unselects a region of the virtual display.

Parameters

lpRect

A pointer to a region of the display to select. The coordinates must be in display cursor coordinates, not pixels. If this parameter is NULL, any selected text in the display is unselected.

dwOptions

One or more options which specifies how the region will be selected. These options may be combined using a bitwise Or operator. The following values may be used:

Value	Description
NVT_SELECT_DEFAULT	The default selection option. If there is a region of the display already selected, it will be cleared and the new region is selected.
NVT_SELECT_CLIPBOARD	Copy the selected text to the clipboard. If this option is not specified, the selected text is buffered and may be copied at a later point.
NVT_SELECT_NOREFRESH	The display is not refreshed when the region is selected. This is useful if the application is going to be selecting multiple regions of the display, or combining more than one region, in order to minimize output to the window.
NVT_SELECT_NOBUFFER	Do not buffer the text in the selected region of the display. The display will show any text as being selected, but it will not be available to be copied by the application. This can be useful if the application is going to select multiple regions and combine them.
NVT_SELECT_COMBINE	If there is already a region of the display that has been selected, the new region is combined with the previous region, selecting all of the text.
NVT_SELECT_UNSELECT	Unselect the specified region of the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

CopySelectedText, GetSelectedText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetAttributes Method

```
UINT SetAttributes(
   UINT nAttributes
);
```

The **SetAttributes** method sets the current display attributes.

Parameters

nAttributes

An unsigned integer value which specifies the new display attributes. This may be one or more of the following values:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

Return Value

If the method succeeds, the return value is the previous display attributes. If the method fails, the return value is 0xFFFFFFFF.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetAttributes, GetCell

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetBackColor Method

```
BOOL SetBackColor(
   COLORREF rgbBackground
);
```

The **SetBackColor** sets the background color used by the virtual display.

Parameters

rgbBackground

The background color specified as a 32-bit RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method sets the background color for normal, reverse, bold and reverse-bold text attributes. To set the background color for a specific attribute, use the **SetColor** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetBackColor, GetColor, GetForeColor, SetBoldColor, SetColor, SetForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetBoldColor Method

```
BOOL SetBoldColor(
   COLORREF rgbBold
);
```

The **SetBoldColor** sets the color used by the virtual display to display text with the bold attribute enabled.

Parameters

rgbBold

The bold attribute color specified as a 32-bit RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method sets the foreground color for the bold and reverse-bold text attributes. To set the color for a specific attribute, use the **SetColor** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetColor, SetBackColor, SetColor, SetForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetCell Method

```
BOOL SetCell(
   INT xPos,
   INT yPos,
   TCHAR nChar
   UINT nAttributes
);

BOOL SetCell(
   TCHAR nChar
   UINT nAttributes
);
```

The **SetCell** method sets the value of the specified cell in the virtual display.

Parameters

xPos

An integer value which specifies the cell column in the virtual display. If this argument is omitted or a value of -1 is specified, the cell at the current cursor position will be used.

yPos

An integer value which specifies the cell row in the virtual display. If this argument is omitted or a value of -1 is specified, the cell at the current cursor position will be used.

nChar

The character that should be displayed at the specified location in the virtual display

nAttributes

An unsigned integer which specifies the attributes for the character in the virtual display. The attributes may be one or more of the following values, combined using a bitwise Or operator:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

Note that certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero. Failure typically indicates that the virtual display has not been created, or that the row and/or column values are outside of the bounds of the display.

Remarks

The **SetCell** method is used to modify a specific character cell in the virtual display, changing both the character and the attributes for that cell. Unlike the higher level methods such as **Write** which process character strings and escape sequences, the **GetCell** and **SetCell** methods allow direct, low-level access to the virtual display in memory. After one or more cells are modified using this function, you should call the **Refresh** method to redraw the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetAttributes, GetCell, Refresh, SetAttributes, Write

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetColor Method

```
BOOL SetColor(
   UINT nColorIndex,
   COLORREF dwColor,
   BOOL bForeground
);
```

The **SetColor** method sets the colors used by the virtual display. Each display has a color table which specifies the foreground and background colors used when displaying text with a specific attribute.

Parameters

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Value	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

dwColor

The RGB color value that will be used.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetColor, SetBackColor, SetBoldColor, SetForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetColorMap Method

```
BOOL SetColorMap(
   COLORREF *lpColor,
   INT nColors
);
```

The **SetColorMap** method modifies the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

lpColor

A pointer to an array of COLORREF values which specifies the values to be used by the emulator when setting a color attribute. If this value is NULL, the default color table will be loaded.

nColors

The number of colors which are stored in the array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid or the number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **SetColorMap** method is useful for changing what values are being used when a color attribute it set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
Black	RGB(0,0,0)			
Red	000000A0h	RGB(160,0,0)		
Green	0000A000h	RGB(0,160,0)		
Yellow	0000A0A0h	RGB(160,160,0)		
Blue	00A00000h	RGB(0,0,160)		
Magenta	00A000A0h	RGB(160,0,160)		
Cyan	00A0A000h	RGB(0,160,160)		
White	00E0E0E0h	RGB(224,224,224)		
Gray	00C0C0C0h	RGB(192,192,192)		
Light Red	008080FFh	RGB(255,128,128)		
Light Green	0090EE90h	RGB(144,238,144)		
Light Yellow	00C0FFFFh	RGB(255,255,192)		
Light Blue	00E6D8ADh	RGB(173,216,230)		

Light Magenta	00FFC0FFh	RGB(255,192,255)
Light Cyan	00FFFFE0h	RGB(224,255,255)
High White	00FFFFFFh	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

The **SetColorMap** method is used to modify the actual color displayed by the emulator. For example, if the emulator processes an escape sequence which sets the current foreground color to white, the actual color displayed could be changed to light green. Passing a NULL pointer as the second parameter restores the original color map back to the default values. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if (pDisplay->GetColorMap(rgbColor, 16) )
{
     rgbColor[7] = rgbColor[15];
     pDisplay->SetColorMap(rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetColor, GetColorMap, GetTextColor, SetColor, SetTextColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetCursorPos Method

```
BOOL SetCursorPos(
   INT nCursorX,
   INT nCursorY
);
```

The **SetCursorPos** method sets the current cursor column and row position on the virtual display.

Parameters

nCursorX

New cursor column position. If this value is greater than the maximum number of columns, the current position is set to the last column on the display. The first column on the display is zero.

nCursorY

New cursor row position. If this value is greater than the maximum number of rows, the current position is set to the last row on the display. The first row on the display is zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero. Typically this would indicate that the virtual display has not been created.

Remarks

The **SetCursorPos** method sets the current cursor position on the virtual display. If the display is in origin mode (a scrolling region has been set), then the cursor row position is bound by the current region.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCursorPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetDC Method

```
BOOL SetDC(
   HDC hDC
);
```

The **SetDC** method sets the device context to be used by the virtual display when updating the window.

Parameters

hDC

Handle to the device context. This parameter may be NULL, any device context that is currently associated with the virtual display will be removed. Note that the application still has the responsibility for deleting the device context, otherwise a handle leak will occur.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

It is not required that the device context be explicitly set by the application. By default, the class will use a device context created using the window attached to the virtual display. If a device context is specified by the application, it must be released when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetDC, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetEmulation Method

```
BOOL SetEmulation(
   UINT nEmulation
);
```

The **SetEmulation** method specifies the type of terminal emulation to be performed by the virtual display.

Parameters

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

Changing the emulation type will not affect the current display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetEmulation, GetMode, Reset, SetMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetFocus Method

```
BOOL SetFocus(
   BOOL bFocus
);
```

The **SetFocus** method sets or removes the focus from the virtual display. This method should be called when the display window receives or loses focus.

Parameters

bFocus

A boolean flag which specifies that the display should receive or lose focus.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

When the virtual display receives focus, it updates the current cursor position and displays the caret. When the display loses focus, the caret is hidden. This method should be called in response to the display window receiving the WM_SETFOCUS and WM_KILLFOCUS messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

UpdateCaret

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetFont Method

```
BOOL SetFont(
   HFONT hFont
);

BOOL SetFont(
   LPTSTR lpszFontName,
   INT nFontSize
);
```

The **SetFont** sets the font that will be used when updating the display. The specified font must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

Parameters

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used.

lpszFontName

A null-terminated string which specifies the name of the font that will be loaded. If a NULL pointer or an empty string is passed as the value, the default **Terminal** font will be used.

nFontSize

The point size of the font that will be loaded. If this value is zero, a default point size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

If the previous font was a default font created by the library as the result of a NULL font handle being passed to a method, it will be released. However, if the previous font was created by the application, the **DeleteObject** method must be called to release it.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetFont

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetForeColor Method

```
BOOL SetForeColor(
   COLORREF rgbForeground
);
```

The **SetForeColor** sets the foreground color used by the virtual display.

Parameters

rgbForeground

The foreground color specified as a 32-bit RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method sets the foreground color for the normal and reverse text attributes. To set the foreground color for a specific attribute, use the **SetColor** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetBackColor, GetColor, GetForeColor, SetBackColor, SetBoldColor, SetColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetMappedKey Method

```
BOOL SetMappedKey(
   UINT nMappedKey,
   LPCTSTR lpszKeyBuffer
);
```

The **SetMappedKey** method maps an escape sequence to the specified key.

Parameters

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (method) keys for the current emulation.

lpszKeyBuffer

Pointer to a string which defines the sequence of characters that are to be mapped to the specified key. Passing an empty string or NULL pointer will delete the current sequence mapped to the key and restore the default value, if one has been defined.

Return Value

If the method succeeds, the return value is non-zero and the escape sequence is mapped to the specified key. If the key value is invalid or could not be mapped, then the method will return zero.

Remarks

The **SetMappedKey** method maps an escape sequence to a special key. This method can be used to specify what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 method key). There are a number of default sequences that are mapped to the method and cursor keys, based on the current emulation. Calling this method will override the default sequence for a key, if one has been defined.

The following special keys are defined:

Value	Description
NVT_F1	F1 method key
NVT_F2	F2 method key
NVT_F3	F3 method key
NVT_F4	F4 method key
NVT_F5	F5 method key
NVT_F6	F6 method key
NVT_F7	F7 method key
NVT_F8	F8 method key
NVT_F9	F9 method key
NVT_F10	F10 method key
NVT_F11	F11 method key
NVT_F12	F12 method key
NVT_SF1	Shift F1 method key

NVT_SF2	Shift F2 method key
NVT_SF3	Shift F3 method key
NVT_SF4	Shift F4 method key
NVT_SF5	Shift F5 method key
NVT_SF6	Shift F6 method key
NVT_SF7	Shift F7 method key
NVT_SF8	Shift F8 method key
NVT_SF9	Shift F9 method key
NVT_SF10	Shift F10 method key
NVT_SF11	Shift F11 method key
NVT_SF12	Shift F12 method key
NVT_ENTER	Enter key
NVT_ERASE	Backspace key
NVT_UP	Cursor up key
NVT_DOWN	Cursor down key
NVT_LEFT	Cursor left key
NVT_RIGHT	Cursor right key
NVT_INSERT	Insert key
NVT_DELETE	Delete key
NVT_HOME	Home key
NVT_END	End key
NVT_PGUP	Page up key
NVT_PGDN	Page down key
NVT_APPUP	Up arrow key
NVT_APPDOWN	Down arrow key
NVT_APPLEFT	Left arrow key
NVT_APPRIGHT	Right arrow key
NVT_APPENTER	Keypad enter key
NVT_KEYPAD0	Numeric keypad 0
NVT_KEYPAD1	Numeric keypad 1
NVT_KEYPAD2	Numeric keypad 2
NVT_KEYPAD3	Numeric keypad 3
NVT_KEYPAD4	Numeric keypad 4
NVT_KEYPAD5	Numeric keypad 5

NVT_KEYPAD6	Numeric keypad 6
NVT_KEYPAD7	Numeric keypad 7
NVT_KEYPAD8	Numeric keypad 8
NVT_KEYPAD9	Numeric keypad 9

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMode, GetMappedKey, ResetMappedKeys, SetMode, TranslateMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetMode Method

```
BOOL SetMode(
   UINT nMode,
   BOOL bEnable
);
```

The **SetMode** method sets one or more display modes for the specified virtual display.

Parameters

nMode

The virtual display mode bitmask. This value is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other methods. The following values may be specified.

NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT (horizontal tab) control sequence is processed. By default this mode is

	disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less then the total number of columns in the virtual display. Disabling this mode prevents a horizontal scroll bar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less then the total number of rows in the virtual display. Disabling this mode prevents a vertical scroll bar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Create, GetDisplayInfo, GetMode, SetEmulation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetScrollPos Method

```
BOOL SetScrollPos(
   INT nScrollBar,
   INT nScrollPos
);
```

The **SetScrollPos** method sets the position of the scroll box for the specified scroll bar and redraws the scroll bar to reflect the new position of the scroll box.

Parameters

nScrollBar

Specifies the scroll bar to be set. This parameter can be one of the following values:

Value	Description
SB_HORZ	Sets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Sets the position of the scroll box in the display's standard vertical scroll bar.

nScrollPos

Specifies the new row or column of the scroll box. The position must be within the scrolling range.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method should always be used to change the scroll box position. The **SetScrollPos** method will result in unpredictable behavior if used on the virtual display window.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetScrollPos, Scroll, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetScrollRegion Method

```
BOOL SetScrollRegion(
   INT nTop,
   INT nBottom
);
```

The **SetScrollRegion** method sets the top and bottom rows of the current scrolling region.

Parameters

nTop

The top row of the current scrolling region. If this value is greater than the bottom scrolling region row or the total number of rows in the display, it will be silently adjusted.

nBottom

The bottom row of the current scrolling region. If this value is less than zero or the top scrolling region row, it will be silently adjusted.

Return Value

If the method succeeds, it will return zero. If the handle to the display is invalid, the method will return zero.

Remarks

The **SetScrollRegion** method allows an application to set the current scrolling region for the virtual display. This specifies the region (between the top and bottom rows) in which text will normally scroll. If the display is in origin mode, the cursor cannot be positioned outside of the scrolling region.

The minimum scrolling region that may be defined is two rows. If a scrolling region is specified that is less than two rows, the method will fail and the current scrolling region will remain unchanged. Specifying values of -1 for both arguments will reset the scrolling region to the default values (the full display).

The DEC STBM escape sequence is used to set or clear the scrolling region of the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetMode, SetMode, SetScrollRegion

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetSize Method

```
BOOL SetSize(
   LPSIZE lpSize
);
```

```
BOOL SetSize(
   INT nColumns,
   INT nRows
);
```

The **SetSize** sets the size of the virtual display.

Parameters

lpSize

Pointer to a SIZE structure which specifies the new size of the virtual display. The width should be specified in columns and the height should be specified in rows.

nColumns

The number of columns in the virtual display.

nRows

The number of rows in the virtual display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetDisplayInfo, GetRect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetTextColor Method

```
BOOL SetTextColor(
   COLORREF rgbColor,
   BOOL bForeground
);
```

The **SetTextColor** method changes the current foreground or background text color.

Parameters

rgbColor

A color value which specifies the current foreground or background text color. The RGB macro can be used to specify the red, green and blue values for the color.

bForeground

A boolean value which determines if the foreground or background color is changed. A value of TRUE indicates that the foreground color should be changed, otherwise the background color is changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. Failure indicates that the handle to the display is invalid or that the current display does not support color text attributes.

Remarks

This method is used to change the current foreground or background color, as determined by the text attribute. Note that changing the current foreground or background text color does not affect the virtual display color table. To change how color attributes are mapped to an RGB color value, use the **SetColorMap** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetColorMap, GetTextColor, SetColorMap

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::SetWindow Method

```
BOOL SetWindow(
   HWND hWnd
);
```

The **SetWindow** method sets the window used by the virtual display.

Parameters

hWnd

Handle to the window that will be used by the virtual display. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Create, GetDC, GetWindow, Reset, SetDC

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::TranslateMappedKey Method

```
BOOL TranslateMappedKey(
   UINT nKey,
   UINT nFlags,
   UINT *lpnMappedKey,
   LPTSTR lpszKeyBuffer,
   UINT cchBuffer
);

BOOL TranslateMappedKey(
   UINT nKey,
   UINT nFlags,
   UINT *lpnMappedKey,
   CString& strKeyBuffer
);
```

The **TranslateMappedKey** method translates a virtual key press to an escape sequence based on the current terminal emulation that has been selected.

Parameters

nKey

The virtual key code for the specified key.

nFlags

The scan code, key-transition code, previous key state, and context code for the specified key.

lpnMappedKey

A pointer to an unsigned integer which will contain the index into the keymap table when the method returns. This is the same value used with the **GetMappedKey** and **SetMappedKey** method. If the index into the keymap is not required, this parameter can be NULL.

lpszKeyBuffer

Address of the buffer to receive the escape sequence mapped to the specified key. If the mapped key string is not required, this parameter can be NULL.

cchBuffer

The maximum number of characters that can be copied into the key buffer, including the terminating null character. If the ***lpszKeyBuffer*** parameter is NULL, this value must be zero.

Return Value

If the virtual key can be mapped to an escape sequence, the method will return a non-zero value. If the key is not mapped, or one of the arguments is invalid, the method will return zero.

Remarks

The **TranslateMappedKey** method allows an application to map a virtual key code to an escape sequence that is appropriate for the type of terminal that is being emulated. For example, it will return the escape sequence for the F1 method key when passed the VK_F1 key value. This method should be called when the WM_KEYDOWN message is processed by an application so that it may send the correct sequence to the server.

This method should only be called in response to a keyboard message such as WM_KEYDOWN. To determine if a specific key has been mapped to an escape sequence, use the **GetMappedKey** method.

Example

```
case WM_KEYDOWN:
    /*
     * If the Num Lock key is pressed, then set the terminal into application
     * keypad mode. This will change how the TranslateMappedKey method will
     * translate the keypad keys.
     *
     * Note that the terminal may also be placed into application keypad mode
     * if emulating a DEC VT terminal and the DECNKM escape sequence is sent
     * by the host.
     */
    if (wParam == VK_NUMLOCK)
        pDisplay->SetMode(NVT_MODE_APPKEYPAD, !(GetKeyState(VK_NUMLOCK) & 1));
    else
    {
        BOOL bMapped;
        UINT nMappedKey;
        TCHAR szKey[128];

        bMapped = pDisplay->TranslateMappedKey(wParam, HIWORD(lParam),
                                               &nMappedKey, szKey, 128);

        if (bMapped)
            pTelnet->Write(szKey);
    }
    break;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetMode, GetMappedKey, ResetMappedKeys, SetMode, SetMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Update Method

```
BOOL Update();
```

The **Update** method updates the window attached to the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method should be called when the display window receives a WM_PAINT message. If the **SetDC** method has not been called to explicitly set the display device context, this method will acquire one for the display window. In this case, the application should not create device context for the window or call the **BeginPaint** and **EndPaint** methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

Refresh, Resize, SetDC, SetScrollPos, SetWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::UpdateCaret Method

```
BOOL UpdateCaret();
```

The **UpdateCaret** method updates the position of the caret in the display window to the current cursor position.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

SetCursorPos, SetMode, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::Write Method

```
BOOL Write(
   LPBYTE lpBuffer,
   INT cbBuffer
);

BOOL Write(
   LPCTSTR lpszBuffer,
   INT cbBuffer
);

BOOL Write(
   TCHAR nChar,
   INT nCount
);
```

The **Write** method writes the contents of the specified buffer to the virtual display.

Parameters

lpBuffer

Pointer to the buffer which contains the data to be written to the virtual display. This may also specify a string which contains the characters to be written to the display.

cbBuffer

Number of bytes to write to the display. If the buffer is a string, this value may be -1, in which case all characters up to the terminating null character will be written to the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method writes the data at the current cursor location. Control characters are recognized by this method and processed accordingly. If ANSI emulation is enabled, embedded escape sequences will also be parsed and processed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

GetText, Refresh, UpdateCaret, Update

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation Data Structures

- NVTDISPLAYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NVTDISPLAYINFO

The **NVTDISPLAYINFO** structure contains information about the virtual terminal display.

```
typedef struct _NVTDISPLAYINFO {
   HWND    hWnd;
   HFONT   hFont;
   INT     xPos;
   INT     yPos;
   INT     cxClient;
   INT     cyClient;
   INT     cxChar;
   INT     cyChar;
   INT     nScrollCol;
   INT     nScrollRow;
   INT     nMaxScrollCol;
   INT     nMaxScrollRow;
} NVTDISPLAYINFO, *LPNVTDISPLAYINFO;
```

Members

hWnd

The handle to the terminal emulation display window.

hFont

The handle to the current font.

xPos

The current display x coordinate.

yPos

The current display y coordinate.

cxClient

The width of the client window in pixels.

cyClient

The height of the client window in pixels.

cxChar

The width of the current font in pixels.

cyChar

The height of the current font in pixels.

nScrollCol

The current horizontal scrolling column.

nScrollRow

The current vertical scrolling row.

nMaxScrollCol

The maximum horizontal scrolling column.

nMaxScrollRow

The maximum vertical scrolling row.

Remarks

This structure is populated by the **GetDisplayInfo** method to provide information about the current state of the virtual terminal. Member of this structure should be modified directly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

Create, GetDisplayInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulator Control Sequences

Terminal Control Sequences

<ESC>c	Reset display to initial state
<ESC>8	Display alignment test

Cursor Control Sequences

<ESC>D	Move cursor down to next line
<ESC>E	Move cursor to first column and down one line
<ESC>M	Move cursor up one line
<ESC>7	Save cursor position, attributes and colors
<ESC>8	Restore saved cursor position, attributes and colors
<ESC>[*n*A	Move cursor up *n* lines
<ESC>[*n*B	Move cursor down *n* lines
<ESC>[*n*C	Move cursor forward *n* spaces
<ESC>[*n*D	Move cursor backward *n* spaces
<ESC>[*n*E	Move cursor to beginning of line, down *n* lines
<ESC>[*n*F	Move cursor to beginning of line, up *n* lines
<ESC>[*x*G	Move cursor to column *x*
<ESC>[*y*;*x*H	Move cursor to line *y*, column *x*
<ESC>[*n*I	Move cursor forward *n* tabstops
<ESC>[*n*Z	Move cursor backwards *n* tabstops
<ESC>[*n*a	Move cursor forward *n* spaces
<ESC>[*y*d	Move cursor to row *y*
<ESC>[*n*e	Move cursor down *n* lines
<ESC>[*y*;*x*f	Move cursor to line *y*, column *x*
<ESC>[s	Save cursor position
<ESC>[u	Return to saved cursor position
<ESC>[x`	Move cursor to column *x*

Attribute and Color Sequence

Select display attributes and color

n Value	Description
0	Reset to default attributes and colors
1	Bold attribute
2	Dim attribute
4	Underline attribute
5	Blink attribute (same as reverse)
7	Reverse attribute
8	Hidden attribute
22	Clear bold attribute
24	Clear underline attribute
25	Clear blink attribute

<ESC>[*n*m	27	Clear reverse attribute
	29	Clear color attributes
	30	Black foreground
	31	Red foreground
	32	Green foreground
	33	Yellow foreground
	34	Blue foreground
	35	Magenta foreground
	36	Cyan foreground
	37	White foreground
	40	Black background
	41	Red background
	42	Green background
	43	Yellow background
	44	Blue background
	45	Magenta background
	46	Cyan background
	47	White background

Character Set Sequences

<ESC>(A	Assign ISO Latin 1 character set to font bank G0
<ESC>(B	Assign United States ASCII character set to font bank G0
<ESC>(0	Assign graphics character set to font bank G0
<ESC>)A	Assign ISO Latin 1 character set to font bank G1
<ESC>)B	Assign United States ASCII character set to font bank G1
<ESC>)0	Assign graphics character set to font bank G1

Erase Sequences

<ESC>[*n*@ Insert ***n*** blank spaces

Erase all or part of the display

<ESC>[*n*J

n Value	Description
0	From current position to end of display
1	From beginning of display to current position
2	Erase the entire display

Erase all or part of a line

<ESC>[*n*K

n Value	Description
0	From current position to end of line
1	From beginning of line to current position
2	Erase the entire line

<ESC>[*n*L Insert ***n*** new blank lines

<ESC>[*n*M	Delete *n* lines from current cursor position
<ESC>[*n*P	Delete *n* characters from current cursor position

Scrolling Sequences

<ESC>[*n*S	Scroll display up *n* lines
<ESC>[*n*T	Scroll display down *n* lines
<ESC>[*n*X	Erase *n* characters from the current position
<ESC>[*y1*;*y2*r	Set scrolling region from lines *y1* to *y2*

Keypad Sequences

<ESC>=	Place keypad into applications mode
<ESC>>	Place keypad into numeric mode

Emulation Option Sequences

<ESC>[?*n*h

Set emulation option

n Value	Description
1	Enable cursor key application mode
2	Enable ANSI escape sequences
5	Reverse foreground and background colors
6	Enable origin mode
7	Enable auto-wrap mode
20	Enable linefeed/newline mode
25	Display caret
66	Place keypad in applications mode

<ESC>[?*n*1

Set emulation option

n Value	Description
1	Disable cursor key application mode
2	Enable VT52 escape sequences
5	Restore foreground and background colors
6	Disable origin mode
7	Disable auto-wrap mode
20	Disable linefeed/newline mode
25	Hide caret
66	Place keypad in numeric mode

Console Escape Sequences

<ESC>[=*n*A	Set the overscan color (ignored)
<ESC>[=*n1*;*n2*B	Set bell sound (parameters ignored)
<ESC>[=*n1*;*n2*C	Set the caret size

<ESC>[=*n*D

Set background color intensity

n Value	Description
0	Decrease background color intensity

	1	Increase background color intensity
<ESC>[=*n*E	Set blink vs. bold attribute (ignored)	
<ESC>[=*n*F	Set normal foreground color	
<ESC>[=*n*G	Set normal background color	
<ESC>[=*n*H	Set reverse foreground color	
<ESC>[=*n*I	Set reverse background color	
<ESC>[=*n*J	Set graphics foreground color	
<ESC>[=*n*K	Set graphics background color	

n Value	Description
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	High White

Control Character Sequences

<CTL>G	Ring audible bell, if enabled
<CTL>H	Move cursor one character backwards
<CTL>I	Move cursor forward to next tabstop
<CTL>J	Move cursor down to next line
<CTL>M	Move cursor to beginning of line
<CTL>N	Select G1 character set
<CTL>O	Select G0 character set
<CTL>Z	Abort current escape sequence
<DEL>	Erase and move cursor one character backwards

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Class Library

Send text messages to a mobile communications device using a gateway service.

Reference

- Class Methods
- Data Structures
- Error Codes

Library Information

Class Name	CTextMessage
File Name	CSTXTV12.DLL
Version	12.0.3024.1286
LibID	B8830D80-2618-4202-9C9C-9C3AD511E9D0
Import Library	CSTXTV12.LIB
Dependencies	None

Overview

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The **CTextMessage** class provides methods that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

This class has been designed to assist developers in sending text message notifications as part of their application. For example, it can be used to enable your software to automatically send notifications when a specific event occurs, such as an error condition. This class is not designed to be used with software that will send out a large number of text messages to many users, and there are limitations on the number of messages that may be sent to different phone numbers over a short period of time. Because many recipients must pay a fee for each text message they receive, text messages should only be sent to those who explicitly request them.

Applications should use an alternate method for sending text messages. Most mobile carriers have shut down their email-to-SMS gateways, making this method of sending text messages unreliable or unavailable. It is recommended that you transition to dedicated SMS service providers, which deliver messages over HTTPS-based APIs instead of email.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Class Methods

Class	Description
CTextMessage	Constructor which initializes the current instance of the class
~CTextMessage	Destructor which releases resources allocated by the class
Method	Description
DisableRelay	Disable the relaying of messages through another mail server
DisableTrace	Disable logging of network function calls
EnableRelay	Enable the relaying of messages through another mail server
EnableTrace	Enable logging of network function calls to a text file
EnumProviders	Enumerate the available wireless service providers
GetAddress	Return the email address associated with the specified phone number.
GetErrorString	Return a description for the specified error code
GetFirstProvider	Return information about the first supported wireless service provider
GetGateway	Return information about the gateway server for the specified phone number
GetLastError	Return the last error code
GetNextProvider	Return information about the next supported wireless service provider
GetProvider	Return information about the wireless service provider for the specified phone number
GetTimeout	Return the amount of time until an operation times out
IsInitialized	Determine if the class has been successfully initialized
IsRelaying	Determine if relaying through another mail server has been enabled
SendMessage	Send a text message to the specified mobile device
SetLastError	Set the last error code
SetTimeout	Set the amount of time to wait before an operation times out
ShowError	Display a message box with a description of the specified error

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::CTextMessage Method

```
CTextMessage();
```

The **CTextMessage** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **SmsInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CTextMessage, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::~CTextMessage

```
~CTextMessage();
```

The **CTextMessage** destructor releases resources allocated by the current instance of the **CTextMessage** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CTextMessage** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

CTextMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::DisableRelay Method

```
VOID DisableRelay();
```

The **DisableRelay** method disables the relaying of text messages through another mail server.

Parameters

None.

Return Value

None.

Remarks

The **DisableRelay** method sets an internal flag that specifies that messages should not be relayed through another mail server. When using the default SMTP service, this means that the library will attempt to send the message directly to the gateway used by the wireless service provider. Relaying is only enabled when the **EnableRelay** method is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableRelay, IsRelaying, SendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::EnableRelay Method

```
BOOL EnableRelay(
   LPCTSTR lpszHostName,
   UINT nHostPort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   DWORD dwOptions,
);

BOOL EnableRelay(
   LPCTSTR lpszHostName,
   UINT nHostPort,
);

BOOL EnableRelay();
```

The **EnableRelay** method enables the relaying of messages through another mail server.

Parameters

lpszHostName

A null-terminated string that specifies the hostname or IP address of the mail server that the messages will be relayed through. If this parameter is NULL or an empty string, then message relaying will be disabled.

nHostPort

An integer value that specifies the port number that should be used to establish the connection with the mail server. If this value is zero, then the default SMTP port will be used.

lpszUserName

A null-terminated string that specifies the user name that will be used to authenticate the session with the mail server. If the server does not require authentication, this parameter can be NULL or an empty string.

lpszPassword

A null-terminated string that specifies the password that will be used to authenticate the session with the mail server. If the server does not require authentication, this parameter can be NULL or an empty string.

dwOptions

An integer value which specifies one or more options.

Value	Description
SMS_OPTION_NONE	No additional options specified. This is the default value.
SMS_OPTION_SECURE	This option specifies that TLS will be used to establish a secure, encrypted connection with the mail server. For some servers, it may be required to connect to them securely and this option will be enabled automatically.

Return Value

If the method succeeds and relaying is enabled, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When a text message is sent using the SMTP service, the default action is to attempt to connect directly to the wireless service provider's gateway server. However, many residential Internet service providers (ISPs) do not permit their customers to connect to third-party mail servers and will block the outbound connection. Some wireless service providers may also reject messages that originate from residential IP addresses.

To resolve this issue, the developer should allow the user to specify an alternate mail server that will relay the message to the wireless service provider. For residential users, this will typically be the mail server provided by their ISP. For business users, this will usually be their corporate mail server. The **EnableRelay** method is used to specify the connection information for the mail server and the **SendMessage** method will relay messages through that server.

Calling the **EnableRelay** method without any arguments will enable relaying through the mail server specified by a previous call to the method. The **EnableRelay** method must be called at least once with arguments to enable relaying. To temporarily disable relaying, call the **DisableRelay** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableRelay, IsRelaying, SendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::EnumProviders Method

```
INT EnumProviders(
   LPSMSPROVIDER lpProviders,
   INT nMaxProviders,
);
```

The **EnumProviders** method enumerates the supported wireless service providers and populates an array of **SMSPROVIDER** structures.

Parameters

lpProviders

A pointer to an array of SMSPROVIDER structures that will be populated with information about each service provider. If this parameter is NULL, the method will return the number of service providers that are known.

nMaxProviders

An integer value which specifies the maximum number of service providers that may be enumerated by this method. If this parameter is zero, the method will return the number of service providers that are known. If the *lpProviders* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of known service providers. If the method fails, the return value is SMS_ERROR. To get extended error information, call **GetLastError**. If the *lpProviders* parameter is not NULL and the *nMaxProviders* parameter indicates the array is not large enough to store all of the provider information, this method will fail with an error indicating that the buffer is too small.

Remarks

The **EnumProviders** method is used to enumerate all of the supported wireless service providers, populating an array of **SMSPROVIDER** structures that contains information about each provider, such as their name, domain, region of the country they service and the maximum message size they will accept. Typically this would be used to update a user interface control such as a listbox or drop-down combobox, enabling a user to select a preferred service provider.

The **GetFirstProvider** and **GetNextProvider** methods offer an alternative way to enumerate the available service providers.

To obtain information about a single service provider, use the **GetProvider** method.

Example

```
SMSPROVIDER smsProviders[MAXPROVIDERS];

INT nProviders = pTextMessage->EnumProviders(smsProviders, MAXPROVIDERS);
if (nProviders == SMS_ERROR)
{
    pTextMessage->ShowError();
}
else
{
    for (INT nIndex = 0; nIndex < nProviders; nIndex++)
        pComboBox->AddString(smsProviders[nIndex].szName));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstProvider, GetGateway, GetNextProvider, GetProvider, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetAddress Method

```
BOOL GetAddress(
   LPCTSTR lpszPhoneNumber,
   LPTSTR lpszAddress,
   INT nMaxLength,
);

BOOL GetAddress(
   LPCTSTR lpszPhoneNumber,
   CString& strAddress
);
```

The **GetAddress** method returns the email address associated with the specified phone number.

Parameters

lpszPhoneNumber

A null-terminated string that specifies the phone number of the mobile device.

lpszAddress

A null-terminated string that will contain the email address associated with the specified phone number when the method returns. If MFC or ATL is being used, an alternate version of this function will return the address in a **CString** type variable.

nMaxLength

The maximum number of characters that may be copied into the string buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetAddress** method is used to determine the email address associated with a mobile device. Sending an email message to this address will forward that message to the device as a text message. Internally, this method calls **GetGateway** and returns the address specified in the SMSGATEWAY structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetGateway, GetProvider, SMSGATEWAY

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetFirstProvider Method

```
BOOL GetFirstProvider(
   LPSMSPROVIDER lpProvider
);
```

The **GetFirstProvider** method returns information about the first supported wireless service provider.

Parameters

lpProvider

A pointer to an SMSPROVIDER structure that will contain information about the service provider. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **GetFirstProvider** method is used in conjunction with **GetNextProvider** to enumerate all of the supported wireless providers available to the client. These two methods can be used as an alternative to the **EnumProviders** method.

Example

```
SMSPROVIDER smsProvider;

BOOL bResult = pTextMessage->GetFirstProvider(&smsProvider);
while (bResult)
{
    pListBox->AddString(smsProvider.szName);
    bResult = pTextMessage->GetNextProvider(&smsProvider);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumProviders, GetGateway, GetNextProvider, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetGateway Method

```
BOOL GetGateway(
   LPCTSTR lpszPhoneNumber,
   LPCTSTR lpszProvider,
   LPSMSGATEWAY lpGateway
);

BOOL GetGateway(
   LPCTSTR lpszPhoneNumber,
   LPSMSGATEWAY lpGateway
);
```

The **GetGateway** method returns text message service information for a phone number.

Parameters

lpszPhoneNumber

A null-terminated string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpszProvider

A null-terminated string which specifies the preferred service provider for this telephone number. If the preferred service provider is unknown, this parameter can be omitted or specify a NULL pointer and the default provider will be selected.

lpGateway

A pointer to an SMSGATEWAY structure that will contain information about the text message gateway when the method returns. This includes information such as the name of the provider, the server that will accept text messages for this phone number, and the recipient address that should be used. This parameter cannot be NULL.

Return Value

If the method succeeds and relaying is enabled, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetGateway** method returns information about the service provider and mail gateway for a specific phone number, and can be used to determine if a given phone number is assigned to a mobile device capable of receiving text messages. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number along with information about its text message gateway service.

If the ***lpszProvider*** parameter is not NULL, this will identify a preferred provider for the phone number specified. In the United States and Canada, most wireless common carriers are required to provide wireless number portability (WNP) which allows a customer to continue to use their current phone number even if they switch to another service provider. This can result in a situation where a specific phone number is shown as allocated to one provider, but in actuality that user has switched to a different provider. For example, a user may have originally purchased a phone and service with AT&T and then later switched to Verizon, but decided to keep their phone number. In this case, if Verizon was not specified as the preferred provider, the library would attempt to send the message to the

AT&T gateway, since that was the original provider who allocated the phone number.

For most applications, the correct way to handle the situation in which a user may have switched to a different service provider is to allow them to select an alternate service provider in your user interface. For example, you could display a drop-down list of available service providers, populated using the **EnumProviders** method. If they select a preferred provider, then you would pass that value to this method. If they do not, then specify a NULL pointer and the default provider will be selected.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumProviders, GetProvider, SMSGATEWAY

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetNextProvider Method

```
BOOL GetNextProvider(
   LPSMSPROVIDER lpProvider,
);
```

The **GetNextProvider** method returns information about the next supported wireless service provider.

Parameters

lpProvider

A pointer to an SMSPROVIDER structure that will contain information about the service provider. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the last service provider has been enumerated or the method fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **GetNextProvider** method is used in conjunction with **GetFirstProvider** to enumerate all of the supported wireless providers available to the client. These two methods can be used as an alternative to the **EnumProviders** method.

Example

```
SMSPROVIDER smsProvider;

BOOL bResult = pTextMessage->GetFirstProvider(&smsProvider);
while (bResult)
{
    pListBox->AddString(smsProvider.szName);
    bResult = pTextMessage->GetNextProvider(&smsProvider);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumProviders, GetGateway, GetNextProvider, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetProvider Method

```
BOOL GetProvider(
   LPCTSTR lpszPhoneNumber,
   LPSMSPROVIDER lpProvider
);

BOOL GetProvider(
   LPCTSTR lpszPhoneNumber,
   LPTSTR lpszProvider,
   INT nMaxLength
);

BOOL GetProvider(
   LPCTSTR lpszPhoneNumber,
   CString& strProvider
);
```

The **GetProvider** method returns information about the service provider for the specified phone number.

Parameters

lpszPhoneNumber

A null-terminated string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpProvider

A pointer to an SMSPROVIDER structure that will contain information about the service provider for the specified phone number.

lpszProvider

A null-terminated string that will contain the name of the wireless service provider associated with the specified phone number when the method returns. If MFC or ATL is being used, an alternate version of this function will return the address in a **CString** type variable.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string.

Return Value

If the method succeeds and relaying is enabled, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetProvider** method returns information about the service provider associated with a phone number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify

an alternate preferred provider to use when sending the text message. For more information, refer to the **GetGateway** method.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

For a list of all supported wireless service providers, use the **EnumProviders** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumProviders, GetGateway, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is SMS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

EnableRelay, SendMessage, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

CTextMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::IsRelaying Method

```
BOOL IsRelaying();
```

The **IsRelaying** method returns whether or not messages will be relayed through another mail server.

Parameters

None.

Return Value

This method returns a non-zero value if relaying has been enabled, otherwise this method will return zero.

Remarks

For more information about relaying text messages, refer to the **EnableRelay** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

DisableRelay, EnableRelay, SendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::SendMessage Method

```
BOOL SendMessage(
   DWORD dwServiceType,
   LPCTSTR lpszProvider,
   LPCTSTR lpszPhoneNumber,
   LPCTSTR lpszSender,
   LPCTSTR lpszMessage
);

BOOL SendMessage(
   LPSMSSERVICE lpService,
   LPSMSMESSAGE lpMessage
);
```

The **SendMessage** method sends a text message to the specified mobile device.

Parameters

dwServiceType

An unsigned integer value which identifies the type of service being used to send the message. Currently there is only one valid service type, **SMS_SERVICE_SMTP** which sends the message to a mail gateway provided by the wireless service provider. This parameter is included for future expansion when multiple alternative services may be used to send the message.

lpszProvider

A null-terminated string that identifies the wireless service provider that is responsible for the specified phone number. If the service provider is unknown, this parameter can be NULL or an empty string. In that case, the default provider for the phone number will be used.

lpszPhoneNumber

A null-terminated string that specifies the phone number for the mobile device. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpszSender

A null-terminated string which identifies the sender of the message and should specify a valid email address. If the recipient replies to the message, the reply will be sent to this address. This parameter cannot be NULL.

lpszMessage

A null-terminated string that contains the message to be sent to the recipient. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

lpService

A pointer to an SMSSERVICE structure that identifies the messaging service that will be used to send the text message. The default service sends the message through the mail server gateway for the wireless service provider associated with the recipient's phone number. This parameter cannot be NULL.

lpMessage

A pointer to an SMSMESSAGE structure that contains information about the message to be sent,

including the sender, the recipient and the text message itself. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method is used to send a text message to a mobile device. This API is designed to support multiple methods of sending text messages, with the default method sending the message through a server gateway established by the wireless service provider.

SMS_SERVICE_SMTP

This message service sends the message through the wireless service provider's mail gateway using the SMTP protocol. However, it is important to note that many of these gateways will not accept messages from a client that is connected to them using a residential Internet service provider. If the application is being run on a system that uses a residential provider, that service provider may also block outbound connections to all mail servers other than their own. These anti-spam measures typically require that most end-user applications specify a relay mail server rather than submitting the message directly to the wireless provider's gateway.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to the **GetGateway** method.

This service also sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnableRelay, GetGateway, GetProvider, SMSMESSAGE, SMSSERVICE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller.

Parameters

nTimeout

The number of seconds until a blocking operation fails. Setting this parameter to zero will use the default timeout period.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

GetTimeout, EnableRelay, SendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CTextMessage::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Data Structures

- SMSGATEWAY
- SMSMESSAGE
- SMSPROVIDER
- SMSSERVICE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSGATEWAY

This structure contains information about a text message gateway server and the recipient.

```
typedef struct _SMSGATEWAY
{
    INT     nGatewayId;
    INT     nCountryCode;
    INT     nAreaCode;
    INT     nExchange;
    INT     nMessageLength;
    DWORD   dwReserved;
    TCHAR   szProvider[SMS_MAXPROVIDERNAMELEN];
    TCHAR   szDomain[SMS_MAXDOMAINNAMELEN];
    TCHAR   szServer[SMS_MAXMAILSERVERLEN];
    TCHAR   szAddress[SMS_MAXMAILADDRESSLEN];
}  SMSCHANNEL, *LPSMSCHANNEL;
```

Members

nGatewayId

An integer value which identifies the gateway record. This value is used internally and an application should not depend on the value not changing over time for a specific telephone number. As new area codes are introduced, the provider database will be updated to reflect these changes and that can result in a change to the gateway ID associated with a specific telephone number.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nAreaCode

An integer value which specifies the Numbering Plan Area (NPA) code, commonly known as the area code. For the United States and Canada, area codes are assigned by the North American Numbering Plan Administration (NANPA). In North America, the area code is digits 1-3 for a 10-digit telephone number. This value, along with the exchange, is used to determine which company provides wireless service for a specific telephone number.

nExchange

An integer value which specifies the exchange area. In North America, the exchange is digits 4-6 for a 10-digit telephone number. This value, along with the area code, is used to determine which company provides wireless service for a specific telephone number.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwReserved

A value reserved for internal use.

szProvider

A null-terminated string which identifies the name of the service provider that is associated with the specified telephone number. Note that this value may not represent the actual company that is providing the wireless service.

szDomain

A null-terminated string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

szServer

A null-terminated string which identifies the host name or IP address of the mail server used to accept text messages for the specified service provider. In some cases, a provider may have multiple gateway servers and this value will represent the preferred mail server for the domain.

szAddress

A null-terminated string which contains the complete email address that should be used when sending the text message through the gateway mail server. Different service providers can have slightly different rules about how the address is formatted, but it typically is a combination of the telephone number and the gateway domain name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSMESSAGE

This structure provides information about a text message.

```
typedef struct _SMSMESSAGE
{
    DWORD dwFormat;
    DWORD dwLength;
    DWORD dwFlags;
    DWORD dwReserved;
    LPCTSTR lpszProvider;
    LPCTSTR lpszPhoneNumber;
    LPCTSTR lpszSender;
    LPCTSTR lpszMessage;
}  SMSMESSAGE, *LPSMSMESSAGE;
```

Members

dwFormat

An integer value which specifies the format of the message message. This member is included for future use where a service provider supports multiple message formats based on different versions of the protocol. The default value for this member is SMS_FORMAT_TEXT.

dwLength

An integer value which specifies the length of the message. If this member is zero, the length will be automatically calculated based on the length of the ***lpszMessage*** text that is terminated by a null character. If this value is larger than the actual length of the message text, it will be ignored.

dwFlags

An integer value which specifies one or more message options.

Value	Description
SMS_MESSAGE_DEFAULT	The default value used with standard text messages.
SMS_MESSAGE_URGENT	The text message should be flagged as urgent. For messages that are sent through a mail gateway, this will set the header to indicate that it is a high priority message. Note that service providers handle urgent messages differently and some may ignore the message priority.

dwReserved

Reserved for future use. This value should always be zero.

lpszProvider

A null-terminated string which specifies the name of the preferred wireless service provider responsible for handling the message. If this member is NULL or an empty string, the default provider assigned to the recipient's phone number will be used. This structure member is only used with SMS_SERVICE_SMTP messages and is ignored for other message services.

lpszPhoneNumber

A null-terminated string which specifies the recipient's phone number. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. This structure member cannot be NULL.

lpszSender

A null-terminated string which identifies the sender of the message. For SMS_SERVICE_SMTP messages, this string should be a valid email address. For other services, this string may specify a phone number or shortcode. This structure member cannot be NULL.

lpszMessage

A null-terminated string that contains the message to be sent to the recipient. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SendMessage, SMSSERVICE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSPROVIDER

This structure contains information about a wireless service provider.

```
typedef struct _SMSPROVIDER
{
    INT     nProviderId;
    INT     nCountryCode;
    INT     nRegionCode;
    INT     nMessageLength;
    DWORD   dwFlags;
    DWORD   dwReserved;
    TCHAR   szGuid[SMS_MAXPROVIDERGUIDLEN];
    TCHAR   szName[SMS_MAXPROVIDERNAMELEN];
    TCHAR   szCompany[SMS_MAXCOMPANYNAMELEN];
    TCHAR   szDomain[SMS_MAXDOMAINNAMELEN];
}  SMSPROVIDER, *LPSMSPROVIDER;
```

Members

nProviderId

An integer value which identifies the provider record. This value is used internally and an application should not depend on the value not changing over time for a specific service provider. To uniquely identify a provider, use the ***szGuid*** member of the structure, which is guaranteed not to change as providers are added and removed from the database.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nRegionCode

An integer value which identifies the region that the service provider covers. In North America, each region consists of multiple states and/or provinces. If a provider services multiple regions, this will identify the primary region where they provide coverage.

Value	Description
SMS_REGION_NATIONAL 0	All regions. This region code is used for service providers that have national coverage and do not exclusively provide service within one or more specific geographical regions.
SMS_REGION_NORTH_EAST_ATLANTIC 1	Northeastern Atlantic region which includes Connecticut, Maine, Massachusetts, New Hampshire, New Brunswick, Newfoundland, Nova Scotia, Rhode Island and Vermont.
SMS_REGION_MIDDLE_ATLANTIC 2	Middle Atlantic region which includes New Jersey, New York, Delaware, District of Columbia, Maryland, Pennsylvania, Quebec, Virginia and West Virginia.
SMS_REGION_EAST_NORTH_CENTRAL 3	Northeastern central region which includes Illinois, Indiana, Michigan, Ohio and Wisconsin.

SMS_REGION_SOUTH_ATLANTIC 4	Southern Atlantic region which includes Florida, Georgia, North Carolina and South Carolina.
SMS_REGION_EAST_SOUTH_CENTRAL 5	Southeastern central region which includes Alabama, Kentucky, Mississippi and Tennessee.
SMS_REGION_WEST_NORTH_CENTRAL 6	Northwestern central region which includes Iowa, Kansas, Manitoba, Minnesota, Missouri, Nebraska, North Dakota, Ontario and South Dakota.
SMS_REGION_WEST_SOUTH_CENTRAL 7	Southwestern central region which includes Arkansas, Louisiana, Oklahoma and Texas.
SMS_REGION_MOUNTAIN 8	Mountain region which includes Alberta, Arizona, Colorado, Idaho, Montana, Nevada, New Mexico, Northwest Territories, Saskatchewan, Utah and Wyoming.
SMS_REGION_PACIFIC 9	Pacific region which includes Alaska, British Columbia, California, Hawaii, Oregon, Washington and Yukon.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwFlags

An unsigned integer value which specifies one or more flags that provides additional information about the service provider. This value is constructed by using a bitwise operator with any of the following constants:

Value	Description
SMS_PROVIDER_DEFAULT 0	A standard service provider. Typically this means that customers have a service contract for their mobile device and pay monthly access and service charges.
SMS_PROVIDER_PREPAID 1	A service provider that offers pre-paid calling cards or fixed month-to-month payments that do not require long-term service contracts.

dwReserved

A value reserved for internal use.

szGuid

A null-terminated string which uniquely identifies the service provider. The string is in a standard format used for globally unique identifiers (GUIDs) and is guaranteed to not change for the service provider it has been assigned to.

szName

A null-terminated string which specifies the name of service provider. Note that this value may not represent the actual company that is providing the wireless service.

szCompany

A null-terminated string which specifies the name of the company associated with the service provider. This may be the same as name of the service provider itself or it may be the name of a parent company that owns the service provider.

szDomain

A null-terminated string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSSERVICE

This structure provides information about the service used to send a text message.

```
typedef struct _SMSSERVICE
{
    DWORD dwServiceType;
    DWORD dwAuthType;
    DWORD dwVersion;
    DWORD dwTimeout;
    DWORD dwOptions;
    DWORD dwReserved;
    LPCTSTR lpszResource;
    LPCTSTR lpszAccount;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
}  SMSSERVICE, *LPSMSSERVICE;
```

Members

dwServiceType

An integer value which identifies the type of service that will be used to send the message. This member can be one of the following values:

Value	Description
SMS_SERVICE_SMTP	The text message will be sent through the mail gateway for the specified service provider. This service uses SMTP to submit the message for delivery, either directly to the server provider's mail gateway server or through a relay server. This is the default service type.

dwAuthType

An integer value which identifies the type of authentication used with the service. This member can be one of the following values:

Value	Description
SMS_AUTH_DEFAULT	The default authentication method for this service type should be used. Most applications should use this value unless a service type provides multiple authentication methods.
SMS_AUTH_USERNAME	The service requires authentication using a username and password. This value can be used with an SMTP service that requires user authentication and is typically needed when using a mail server relay. For the SMS_SERVICE_SMTP service, this is the default authentication method.

dwVersion

An integer value which identifies the interface version for the service being used. This member is included for future use where a service may support multiple versions of their interface and should normally be set to the value SMS_VERSION_DEFAULT.

dwTimeout

An integer value which specifies the amount of time in seconds that a function will wait for a response from the service. If this value is zero, a default timeout period of 20 seconds will be used.

If the service does not respond within this time period, the function will fail.

dwOptions

An integer value which specifies one or more options.

Value	Description
SMS_OPTION_NONE	No additional options for the service.
SMS_OPTION_SECURE	This option specifies that TLS will be used to establish a secure, encrypted connection with the service. For some services, it may be required to connect to them securely and this option will be enabled automatically.

dwReserved

Reserved for future use. This value should always be zero.

lpszResource

A null-terminated string that specifies a resource for the service. Typically this will be either a fully qualified domain name or a URL. For gateways using SMTP, this string should identify the mail server. An alternate port number can also be specified by appending it to the hostname, separated by a colon. For example, **smtp.company.com:587** would connect to the server on port 587. If you are specifying an IPv6 address with an alternate port number, the address must be enclosed in brackets. For services where a domain name or resource URL is not required, this member will be ignored and can be NULL.

lpszAccount

A null-terminated string that specifies an account name or identifier. Some service providers may require a unique account name or a token (application ID) in conjunction with other credentials. This member is not used with mail gateways and ignored if the service type is SMS_SERVICE_SMTP. If no account name is required for session authentication, this member can be NULL.

lpszUserName

A null-terminated string that specifies a user name to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid user name. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "username" with their documentation; this member will always specify the first of a pair of authentication tokens.

lpszPassword

A null-terminated string that specifies the password used to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid password. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "password" with their documentation; this member will always specify the second of a pair of authentication tokens.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SendMessage, SMSMESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage Application Storage Class Library

The CWebStorage class provides data storage for applications.

Reference

- Class Methods
- Constants
- Data Structures
- Error Codes

Library Information

Class Name	CWebStorage
File Name	CSWEBV12.DLL
Version	12.0.3024.1286
LibID	ECC96063-0D95-4378-A747-E0040B8618F1
Import Library	CSWEBV12.LIB
Dependencies	None

Overview

The CWebStorage class enables an application to store and manage data remotely. These methods use secure services provided by SocketTools API servers and do not require third-party party APIs or accounts with other cloud service providers.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most

commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage Class Methods

Class	Description
CWebStorage	Constructor which initializes the current instance of the class
~CWebStorage	Destructor which releases resources allocated by the class
Method	Description
Cancel	Cancel a storage operation that is currently in progress
CloseStorage	Close the storage container and release resources allocated for the client session
CompareFile	Compare the contents of a stored object with a local file
CompareData	Compare the contents of a stored object with a memory buffer
CompareText	Compare the contents of a stored object with a string
CopyObject	Copy a storage object to a new location and optionally rename the object label
DeleteObject	Delete an existing storage object
DownloadFile	Download the contents of a stored object to a local file
DisableTrace	Disable logging of method calls to the trace log file
EnableTrace	Enable logging of method calls to a log file
EnumObjects	Enumerate all storage objects that match the specified label or content type
GetAccountId	Return the current web services account identifier
GetData	Download the contents of a storage object to a memory buffer
GetErrorString	Return a description for the specified error code
GetFile	Download the contents of a storage object to a local file
GetFirstApplication	Return information about the first registered application
GetFirstObject	Return information about the first object that matches search criteria
GetHandle	Return the storage handle used by this instance of the class
GetLastError	Return the last error code
GetNextApplication	Return information about the next registered application
GetNextObject	Return information about the next object that matches search criteria
GetObjectInformation	Retrieve the metadata for the specified storage object
GetObjectSize	Return the size of the specified storage object
GetObjectTime	Return the creation and last modification time for the specified storage object
GetStorageQuota	Return quota limits assigned to your storage account
GetStorageId	Return the returns the current storage container ID
GetTimeout	Get the number of seconds until a storage operation times out
GetTransferStatus	Return status information about the progress of a data transfer
IsBlocking	Determine if the session waiting for a response from the storage server

IsConnected	Determine if the client is connected to the storage server
IsInitialized	Determine if the class has been successfully initialized
MoveObject	Move a storage object to a new location and optionally rename the object label
OpenStorage	Open a storage container and return a handle for the client session
PutData	Upload the contents of a memory buffer and return information about the new object
PutFile	Upload the contents of a local file and return information about the new object
RegisterAppId	Register a new application identifier used to store and retrieve data
RegisterEvent	Register an event handler to receive notifications for the session
RenameObject	Change the label associated with a storage object
ResetStorage	Resets the application storage container and deletes all stored objects
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until a storage operation times out
ShowError	Display a message box with a description of the specified error
UnregisterAppId	Unregister the application identifier and delete all associated storage objects
UnregisterEvent	Unregister an event handler and stop receiving notifications for the session
UploadFile	Upload the contents of a local file
ValidateAppId	Validate the specified application identifier
ValidateLabel	Check the specified string to ensure it is a valid object label

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::CWebStorage

```
CWebStorage();
```

The **CWebStorage** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only method on the development system and cannot be redistributed.

The constructor calls the **WebInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors.

You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CWebStorage, IsInitialized, WebInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::~CWebStorage

```
~CWebStorage();
```

The **CWebStorage** destructor releases resources allocated by the current instance of the **CWebStorage** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CWebStorage** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CWebStorage, IsConnected, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::Cancel Method

```
BOOL Cancel();
```

The **Cancel** method cancels the current data transfer in progress.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Cancel** method will cancel the current data transfer and abort the connection to the storage server. This method will fail if an active data transfer (either an upload or download) is not in progress.

This would typically be used within an event handler to cancel an operation as the contents of an object are being read or written. There is no mechanism to resume a canceled data transfer and this method should only be used when absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

GetTransferStatus, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::CloseStorage Method

```
BOOL CloseStorage();
```

The **CloseStorage** method closes the open storage container.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CloseStorage** method must be called after all operations using the storage container have completed. The access token granted to the application will be released and the memory allocated for the session cache will be freed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

GetHandle, OpenStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::CompareData Method

```
BOOL CompareData(
   LPCTSTR lpszObjectLabel,
   LPCVOID lpvBuffer,
   DWORD dwLength
);
```

The **CompareData** method compares the contents of a stored object with the data provided by the caller.

Parameters

lpszObjectLabel

A null-terminated string which specifies the label of the object to be compared.

lpvBuffer

A buffer that contains the data to be compared against the storage object.

dwLength

An unsigned integer which specifies the length of the data buffer to be compared.

Return Value

If the method succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CompareData** method performs a binary comparison of the data in the specified buffer with the contents of the storage object on the server. The *dwLength* parameter must match the size of the stored object exactly, or this method will fail. Partial comparisons are not supported by this method.

If you wish to compare the contents of a text object, it is recommended that you use **CompareText**. This method ensures that Unicode text is compared correctly.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareFile, CompareText, GetObjectInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::CompareFile Method

```
BOOL CompareFile(
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszLocalFile
);
```

The **CompareFile** method compares the contents of a stored object with a local file.

Parameters

lpszObjectLabel

A null-terminated string which specifies the label of the object to be compared.

lpszLocalFile

A null-terminated string that specifies the name of the local file. If a path is not specified, the file will be created in the current working directory.

Return Value

If the method succeeds, the return value is a non-zero and the contents of the file matches the stored object. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CompareFile** method performs a binary comparison of the contents of a local file with a stored object on the server. The contents of the file must be identical to the contents of the stored object or the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareData, CompareText, GetObjectInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::CompareText Method

```
BOOL CompareText(
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszObjectText,
   INT cchObjectText
);
```

The **CompareText** method compares the contents of a stored object with the string provided by the caller.

Parameters

lpszObjectLabel

A null-terminated string which specifies the label of the object to be compared.

lpszObjectText

A null-terminated string which contains the text to be compared with the stored object.

cchObjectText

The number of characters in the ***lpszObjectText*** string to be compared. This value may be -1, in which case the string length will be determined by counting the number of characters up to the terminating null.

Return Value

If the method succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CompareText** method performs a text comparison of the characters in the string with the contents of the storage object on the server. The string length must match the amount of text in the stored object exactly, or this function will fail. Partial comparisons are not supported by this function.

Unicode strings will be automatically converted from UTF-16 to UTF-8 encoding in the same way the **PutData** method does when storing text objects.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareFile, CompareData, GetData, GetObjectInformation, PutData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::CopyObject Method

```
BOOL CopyObject(
   LPCTSTR lpszOldLabel,
   LPCTSTR lpszNewLabel,
   DWORD dwStorageType,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **CopyObject** method creates a copy of an existing storage object using a new label.

Parameters

lpszOldLabel

A null-terminated string which specifies the label of the existing storage object to be copied. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A null-terminated string which specifies the name of the new storage object that will be created. This parameter may be NULL or point to an empty string, in which case the label name is not changed. In this case, the ***dwStorageType*** parameter cannot be WEB_STORAGE_DEFAULT.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the value WEB_STORAGE_DEFAULT will be used. One of the following values can be specified:

Value	Description
WEB_STORAGE_DEFAULT	The default storage type. If this value is specified, the new object will be created using the same storage type as the original storage object.
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type

	are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the new storage object. If this information is not required, this parameter may be omitted or a NULL pointer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CopyObject** method is used to create a copy of an existing storage object. It may be used to duplicate an object with a different label, or it may be used to copy the object to a new storage container type. For example, it can copy an object originally created using WEB_STORAGE_USER to a new object stored using WEB_STORAGE_MACHINE.

Copied objects are assigned their own unique ID and are not linked to one another. Any subsequent changes made to the original object will not affect the copied object. Attempting to copy an object to itself or another existing object will result in an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteObject, MoveObject, RenameObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::DeleteObject Method

```
BOOL DeleteObject(
   LPCTSTR lpszObjectLabel
);
```

The **DeleteObject** method deletes an object from the storage container.

Parameters

lpszObjectLabel

A null-terminated string which specifies the label of the object to be deleted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **DeleteObject** method permanently deletes the storage object and its associated data from the server. Deleted objects cannot be recovered by the application. To remove all objects stored in the container, use the **ResetStorage** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CopyObject, MoveObject, RenameObject, ResetStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::DownloadFile Method

```
BOOL DownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject
);

BOOL DownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszAppId,
   DWORD dwStorageType,
   DWORD dwTimeout,
   LPWEB_STORAGE_OBJECT lpObject,
   WEBEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **DownloadFile** method downloads the contents of a storage object and copies it to a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A null-terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, WEB_STORAGE_GLOBAL will be used as the default. One of the following values can be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to

	the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwTimeout

An integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that has been downloaded. This parameter may be omitted or NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback method, see the description of the **WebEventProc** callback method. If this parameter is omitted or NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback method specified by ***lpEventProc***. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **DownloadFile** method downloads the contents of the storage object and stores it in a local file. Unlike the **GetFile** method, it is not required that you explicitly open a storage container using the **OpenStorage** method prior to calling this method.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are downloading a large object and want your application to receive progress updates during the data transfer, provide a pointer to a static callback function as the ***lpEventProc*** parameter. That function will receive event notifications as the data is being downloaded.

Example

```
WEB_STORAGE_OBJECT webObject;

// Download the object from global storage to a local file
if (pStorage->DownloadFile(lpszObjectLabel, lpszLocalFile, &webObject))
{
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be retrieved, display the error
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to retrieve \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, PutFile, UploadFile,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::EnumObjects Method

```
BOOL EnumObjects(
   LPCTSTR lpszMatchLabel,
   LPCTSTR lpszContentType,
   LPWEB_STORAGE_OBJECT& lpObjects,
   DWORD& dwObjects
);

DWORD EnumObjects(
   LPCTSTR lpszMatchLabel,
   LPCTSTR lpszContentType
);
```

The **EnumObjects** method enumerates all storage objects that match the specified label or content type.

Parameters

lpszMatchLabel

A null-terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this value is a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A null-terminated string which specifies the content type of the objects to be enumerated. If this value is a NULL pointer or an empty string, the content type is ignored and all matching objects are returned.

lpObjects

A pointer to an array of WEB_STORAGE_OBJECT structures that will contain information about the enumerated objects. If this parameter is NULL, then no object information is returned.

lpdwObjects

A pointer to an unsigned integer that will contain the number of objects enumerated by this method. If the *lpObjects* parameter points to an array of WEB_STORAGE_OBJECT structures, this parameter must be initialized to the maximum size of the array being passed to the method. If the *lpObjects* parameter is NULL, this value must be initialized to zero, and when the method returns it will contain the number of matching objects.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **EnumObjects** method can be used to enumerate the number of objects that match a given label, content type or both. If a content type is specified, it must be a valid MIME media content type designated using the type/subtype nomenclature. For example, "text/plain" or "image/jpeg". An invalid MIME type will cause the method to fail.

This method can be used in two ways. If the *lpObjects* parameter is NULL and the value passed by reference in the *lpdwObjects* parameter is initialized to zero, the method will return the number of

matching objects in *lpdwObjects*. This can be used to dynamically determine the size of the lpObjects array instead of declaring a fixed-size array in your application.

If the *lpObjects* parameter is not NULL, then the value referenced by *lpdwObjects* must be initialized to the maximum size of the array that *lpObjects* points to. It is important to note that if either parameter is not initialized correctly, it can result in memory corruption and/or an unhandled exception.

There is an alternative version of **EnumObjects** which only returns a count of objects that match the specified content type and/or object label name, without returning information about the objects. This can be used to determine how many objects match the given criteria prior to requesting the object metadata.

The **GetFirstObject** and **GetNextObject** methods can be used to iterate through all matching storage objects without allocating memory to store all of the matching objects. Using **GetFirstObject** and **GetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

Example

```
DWORD dwObjects = 0;
LPWEB_STORAGE_OBJECT lpObjects = NULL;

if (pStorage->EnumObjects(_T("*.pdf"), NULL, lpObjects, dwObjects))
{
    // Print information about each object
    for (DWORD dwIndex = 0; dwIndex < dwObjects; dwIndex++)
    {
        _tprintf(_T("Object: %s\n"), lpObjects[dwIndex].szObjectId);
        _tprintf(_T("Label: %s\n"), lpObjects[dwIndex].szLabel);
        _tprintf(_T("Size: %lu\n"), lpObjects[dwIndex].dwObjectSize);
        _tprintf(_T("Digest: %s\n"), lpObjects[dwIndex].szDigest);
        _tprintf(_T("Content: %s\n"), lpObjects[dwIndex].szContent);

        if (dwObjects > 1 && dwIndex < dwObjects - 1)
            _tprintf(_T("\n"));
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstObject, GetNextObject, GetObjectInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetAccountId Method

```
BOOL GetAccountId(
   LPCTSTR lpszAccountId,
   INT nMaxLength
);

BOOL GetAccountId(
   CString& strAccountId
);
```

The **GetAccountId** method returns the web services account ID associated with the current session.

Parameters

lpszAccountId

A null-terminated string that will contain the account ID when the method returns. This parameter cannot be NULL and must be at least 35 characters in length. An alternate form of the method accepts a **CString** variable which will contain the account ID.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszAccountId* string parameter, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The account ID is a string that uniquely identifies the web services account that is associated with the session. The account ID corresponds with your product serial number and runtime license key, but it is not identical to either of those values.

If you are using an evaluation license, the account ID is temporary and only valid during the evaluation period. After the evaluation period has expired, the account ID is revoked and objects stored using this ID will be deleted. It is not recommended that you store critical application data using an evaluation license.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetStorageId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetData Method

```
BOOL GetData(
   LPCTSTR lpszObjectLabel,
   LPBYTE lpBuffer,
   LPDWORD lpdwLength,
   LPWEB_STORAGE_OBJECT lpObject,
);

BOOL GetData(
   LPCTSTR lpszObjectLabel,
   HGLOBAL *lphgblBuffer,
   LPDWORD lpdwLength,
   LPWEB_STORAGE_OBJECT lpObject,
);

BOOL GetData(
   LPCTSTR lpszObjectLabel,
   LPTSTR lpszBuffer,
   INT nMaxLength,
   LPWEB_STORAGE_OBJECT lpObject,
);

BOOL GetData(
   LPCTSTR lpszObjectLabel,
   CString& strBuffer,
   LPWEB_STORAGE_OBJECT lpObject,
);
```

The **GetData** method retrieves the contents of a storage object and copies it to the memory buffer that is provided. Additional information about the object is optionally returned to the caller.

Parameters

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpBuffer

A pointer to a byte array which will contain the object's data. When this version of the method is called, the buffer must be pre-allocated by the application and large enough to store all of the data or the method will fail. The ***lpdwLength*** parameter must be initialized with the maximum size of the byte array, and will be updated with the actual number of bytes copied into the buffer when the method returns.

lphgblBuffer

A pointer to a HGLOBAL memory handle. When this version of the method returns, the handle will reference a block of memory allocated by the **GlobalAlloc** function and the ***lpdwLength*** parameter will contain the number of bytes copied. To obtain a pointer to the data, the application must call the **GlobalLock** function. It is the responsibility of the application to free the global memory handle when it is no longer needed.

lpszBuffer

A null-terminated string buffer which will contain the object's text when the method returns. When this version of the method is called, the string will be populated with the object contents and terminated with a null character. The ***nMaxLength*** parameter must specify the maximum number of

characters that can be copied into the string, including the terminating null. This version of the function should only be used with text and must never be used with binary data.

strBuffer

A **CString** which will contain the object's text when the method returns. This version of the method requires the application be compiled with MFC or ATL support. This version of the function should only be used with text and must never be used with binary data.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpBuffer*** parameter. If the ***lpBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the ***lpszBuffer*** string buffer provided by the caller. This value must be large enough to contain the entire contents of the object, including the terminating null character.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetData** method is used to retrieve a stored object from the server and copy it into a local buffer. The method may be used in one of several ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the object. In this case, the ***lpBuffer*** parameter will point to the buffer that was allocated and the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is to provide a pointer to an HGLOBAL global memory handle which will contain the file data when the method returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

The third method is to provide a string buffer which will contain the contents of a text object. The ***nMaxLength*** parameter must specify a size large enough to contain the entire object text, including a terminating null character. If the **PutData** method was previously used to store UTF-16 encoded text, you must use a byte array to retrieve that data. This method only recognizes UTF-8 encoded text and considers UTF-16 encoded text to be binary data. This version of the method should never be used to retrieve data from an object that does not contain text.

If you wish to retrieve the contents of an object and store it in a file, use the **GetFile** method.

Additional metadata about the object can be returned in a WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD dwLength = 0;
LPCTSTR lpszObjectLabel = _T("MyAppData");

// Open the default global storage container
if (!pStorage->OpenStorage(WEB_STORAGE_GLOBAL))
{
    _tprintf(_T("Unable to open global storage\n"));
    return;
}

// Return the file data into block of global memory allocated by
// the GlobalAlloc method; the handle to this memory will be
// returned in the hgblBuffer parameter
if (pStorage->GetData(lpszObjectLabel, &hgblBuffer, &dwLength))
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
else
{
    // The GetData method failed
    _tprintf(_T("Unable to retrieve object \"%s\"\n"), lpszObjectLabel);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetObjectSize, GetObjectTime, PutFile, PutData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetFile Method

```
BOOL GetFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **GetFile** method downloads the contents of a storage object and copies it to a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

If you are downloading a large object and want your application to receive progress updates during the data transfer, use the **RegisterEvent** method and provide a pointer to a static callback function that will receive event notifications.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetData, PutFile, PutData, RegisterEvent, UploadFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetFirstApplication Method

```
BOOL GetFirstApplication(
   LPWEB_STORAGE_APPLICATION lpAppInfo
);
```

The **GetFirstApplication** method returns information about the first registered application for the current storage account.

Parameters

lpAppInfo

A pointer to a WEB_STORAGE_APPLICATION structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstApplication** method returns information about the first registered application. It is used in conjunction with the **GetNextApplication** method to enumerate all of the registered application IDs associated with your storage account.

The cached application ID information used by this method is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multithreaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;

if (pStorage->GetFirstApplication(&webApp))
{
    do
    {
        // Print information for each registered application
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);
        _tprintf(_T("\n"));
    }
    while (pStorage->GetNextApplication(&webApp));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetNextApplication, RegisterAppId, UnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetFirstObject Method

```
BOOL GetFirstObject(
   LPWEB_STORAGE_OBJECT lpObject
);

BOOL GetFirstObject(
   LPCTSTR lpszMatchLabel,
   LPWEB_STORAGE_OBJECT lpObject
);

BOOL GetFirstObject(
   LPCTSTR lpszMatchLabel,
   LPCTSTR lpszContentType,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **GetFirstObject** method returns information about the first storage object that matches the specified label or content type.

Parameters

lpszMatchLabel

A null-terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this parameter is omitted, a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A null-terminated string which specifies the content type of the objects to be enumerated. If this parameter is omitted, a NULL pointer or an empty string, the content type is ignored and all matching objects are returned. If a content type is specified, it must be a valid MIME media content type designated using the type/subtype nomenclature.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **GetFirstObject** method returns information about the first object that matches a given label, content type or both. It is used in conjunction with the **GetNextObject** method to enumerate all of the matching objects in the storage container.

This method provides an alternative to the **EnumObjects** method, which populates an array of WEB_STORAGE_OBJECT structures. In some cases, iterating over each object in a loop may be preferred to preallocating memory for an array to store every matching object. Using **GetFirstObject** and **GetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

You cannot intermix calls between **GetFirstObject** and **EnumObjects**. The **EnumObjects** method will

reset the internal object cache for the client session and subsequent calls to **GetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;

if (pStorage->GetFirstObject(&webObject))
{
    do
    {
        // Print information about each object
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);
        _tprintf(_T("Label: %s\n"), webObject.szLabel);
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);
        _tprintf(_T("Content: %s\n"), webObject.szContent);
        _tprintf(_T("\n"));
    }
    while (pStorage->GetNextObject(&webObject));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumObjects, GetNextObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetHandle Method

```
HSTORAGE GetHandle();
```

The **GetHandle** method returns the storage handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_HANDLE will be returned.

Remarks

This method is used to obtain the storage handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Methods set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Method failure is typically indicated by a return value such as FALSE, NULL, INVALID_HANDLE or WEBAPI_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetNextApplication Function

```
BOOL GetNextApplication(
   LPWEB_STORAGE_APPLICATION lpAppInfo
);
```

The **GetNextApplication** method returns information about the next registered application for the current storage account.

Parameters

lpAppInfo

A pointer to a WEB_STORAGE_APPLICATION structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextApplication** method returns information about the next registered application after initially calling the **GetFirstApplication** method. It is used to enumerate all of the registered application IDs associated with your storage account.

The cached application ID information used by this method is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multithreaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;

if (pStorage->GetFirstApplication(&webApp))
{
    do
    {
        // Print information for each registered application
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);
        _tprintf(_T("\n"));
    }
    while (pStorage->GetNextApplication(&webApp));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFirstApplication, RegisterAppId, UnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetNextObject Method

```
BOOL GetNextObject(
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **GetNextObject** method returns information about the next storage object that matches the specified label or content type.

Parameters

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextObject** method returns information about the next object that matches the label and/or content type that was specified by the **GetFirstObject** method. This method may only be called after **GetFirstObject** has been called, otherwise it will fail.

When all matching objects have been returned to the caller, this method will return zero (FALSE) and the **GetLastError** method will return NO_ERROR. Any other error code indicates an underlying problem with the request, such as an invalid parameter passed to the function.

You cannot intermix calls between **GetNextObject** and **EnumObjects**. The **EnumObjects** method will reset the internal object cache for the client session and subsequent calls to **GetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;

if (pStorage->GetFirstObject(&webObject))
{
    do
    {
        // Print information about each object
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);
        _tprintf(_T("Label: %s\n"), webObject.szLabel);
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);
        _tprintf(_T("Content: %s\n"), webObject.szContent);
        _tprintf(_T("\n"));
    }
    while (pStorage->GetNextObject(&webObject));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

EnumObjects, GetFirstObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetObjectInformation Method

```
BOOL GetObjectInformation(
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **GetObjectInformation** method retrieves the metadata for a storage object.

Parameters

lpszObjectLabel

A null-terminated string that specifies the label of the storage object.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter cannot be a NULL pointer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetObjectInformation** method is used to retrieve the metadata for a stored object on the server, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents. This method can also be used as a simple method to determine if the specified object exists without the overhead of requesting the server attempt to retrieve the contents of an object.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

If you are only interested in obtaining the size of a stored object, you can use the **GetObjectSize** method.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **GetStorageQuota** method.

Example

```
WEB_STORAGE_OBJECT webObject;

// Get information about the object
if (pStorage->GetObjectInformation(lpszObjectLabel, &webObject))
{
    // The object exists, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object does not exist, display the error
```

```
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to get information on \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetData, GetObjectSize, GetObjectTime, GetStorageQuota, PutFile, PutData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetObjectSize Method

```
DWORD GetObjectSize(
   LPCTSTR lpszObjectLabel
);

BOOL GetObjectSize(
   LPCTSTR lpszObjectLabel,
   LPDWORD lpdwObjectSize
);
```

The **GetObjectSize** method returns the size of the stored object.

Parameters

lpszObjectLabel

A null-terminated string that specifies the label of the storage object. This parameter cannot be a NULL pointer.

lpdwObjectSize

A pointer to an unsigned integer value that will contain the size of the object. If this parameter is NULL, the parameter is ignored and the method only checks for the existence of an object that matches the specified label.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetObjectSize** method is used to retrieve the size of a stored object on the server. This method can also be used as a simple method to determine if the specified object exists. An object size of zero means the object exists, but currently has no data associated with it.

If the first version of this method is called, where the object size is a return value, you can distinguish between an error and an empty object by calling the **GetLastError** method. If the last error code is NO_ERROR then the object exists, but it has no data associated with it. Any other error code value indicates the reason the object size could not be returned.

If the second version of this method is used and the *lpObjectSize* parameter is not NULL, its value is initialized to zero when the method is called, and updated with the object size when the method returns.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting the size of an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

The **GetObjectInformation** method can be used to obtain the metadata associated with the storage object, including the size, content type, creation date and a SHA-256 digest of the data.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **GetStorageQuota** method.

Example

```
// Check if the object exists
DWORD dwObjectSize = 0;
```

```
if (pStorage->GetObjectSize(lpszObjectLabel, &dwObjectSize))
{
    // The object exists, display the size in bytes
    _tprintf(_T("The size of \"%s\" is %lu bytes\n"), lpszObjectLabel,
dwObjectSize);
}
else
{
    // The object size could not be determined, display the error
    CString strError;

    pStorage->GetErrorString(strError);
    _tprintf(_T("Unable to get the size of \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetData, GetObjectInformation, GetObjectTime, GetStorageQuota, PutData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetObjectTime Method

```
BOOL GetObjectTime(
   LPCTSTR lpszObjectLabel,
   LPFILETIME lpTimeCreated,
   LPFILETIME lpTimeModified
);

BOOL GetObjectTime(
   LPCTSTR lpszObjectLabel,
   LPSYSTEMTIME lpTimeCreated,
   LPSYSTEMTIME lpTimeModified
);
```

The **GetObjectTime** method returns the creation and last modification time of the stored object.

Parameters

lpszObjectLabel

A null-terminated string that specifies the label of the storage object. This parameter cannot be a NULL pointer.

lpTimeCreated

A pointer to a FILETIME structure which will contain the time the storage object was created when the method returns. If this information is not required, the parameter may be NULL. An alternate version of this method accepts a pointer to a SYSTEMTIME structure instead of a FILETIME structure.

lpTimeModified

A pointer to a FILETIME structure which will contain the time the storage object was last modified when the method returns. If this information is not required, the parameter may be NULL. An alternate version of this method accepts a pointer to a SYSTEMTIME structure instead of a FILETIME structure.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetObjectTime** method is used to retrieve the creation and modification times for a stored object on the server. This method can also be used as a simple method to determine if the specified object exists. Each of the LPFILETIME or LPSYSTEMTIME parameters are optional and may be NULL if either the creation or modification times are not needed. If a pointer to a structure is provided, its members will be initialized to zero and then updated with the creation and/or modification time if the method succeeds.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting the size of an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

If you wish to obtain all of the metadata associated with a specific object with a single method call, including the size, content type, creation and modification times, use the **GetObjectInformation** method.

Example

```
// Check if the object exists
FILETIME ftCreated;
FILETIME ftModified;

if (pStorage->GetObjectTime(lpszObjectLabel, &ftCreated, &ftModified))
{
    // The object exists, display the last modification time
    TCHAR szLocalDate[128];
    TCHAR szLocalTime[128];
    FILETIME ftLocalTime;
    SYSTEMTIME stLocalTime;

    FileTimeToLocalFileTime(&ftModified, &ftLocalTime);
    FileTimeToSystemTime(&ftLocalTime, &stLocalTime);

    GetDateFormat(LOCALE_USER_DEFAULT,
                  DATE_SHORTDATE,
                  &stLocalTime,
                  NULL,
                  szLocalDate, 128);

    GetTimeFormat(LOCALE_USER_DEFAULT,
                  TIME_FORCE24HOURFORMAT,
                  &stLocalTime,
                  NULL,
                  szLocalTime, 128);

    _tprintf(_T("The object \"%s\" was lasted modified on %s %s\n"),
lpszObjectLabel, szLocalDate, szLocalTime);
}
else
{
    // The object times could not be determined, display an error
    CString strError;

    pStorage->GetErrorString(strError);
    _tprintf(_T("Unable to get the size of \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetData, GetFile, GetObjectInformation, GetStorageQuota, PutData, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetStorageId Method

```
BOOL GetStorageId(
   LPCTSTR lpszStorageId,
   INT nMaxLength
);

BOOL GetStorageId(
   CString & strStorageId
);
```

The **GetStorageId** method returns the current storage container ID.

Parameters

lpszStorageId

A null-terminated string that will contain the storage ID when the method returns. This parameter cannot be NULL and must be at least 35 characters in length. An alternate version of this method accepts a **CString** parameter if the application is compiled with support for MFC or ATL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszStorageId* string parameter, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The storage ID is a string that identifies the storage container that was opened with the **OpenStorage** method. The storage ID is associated with your account ID and development license and is guaranteed to be a unique value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetAccountId, OpenStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetStorageQuota Method

```
BOOL GetStorageQuota(
   LPWEB_STORAGE_QUOTA lpQuota
);
```

The **GetStorageQuota** method returns quota limits assigned to your development account.

Parameters

lpQuota

A pointer to a WEB_STORAGE_QUOTA structure that will contain information the quota limits for your account when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetStorageQuota** method can be used to determine how much storage space is available to your application. The *ulBytesFree* structure member will tell you how many bytes of storage you have available, and the *dwObjectSize* member will tell you the maximum size of any individual storage object. In addition to the bytes allocated for storage, there is also a limit on the total number of objects that your application may create which is specified by the *dwObjectLimit* member.

Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

Example

```
CWebStorage appStorage;
WEB_STORAGE_QUOTA appQuota;

// Display storage account usage and limits
if (appStorage.GetStorageQuota(&appQuota))
{
     _tprintf(_T("Objects Used:  %lu\n"), appQuota.dwObjects);
     _tprintf(_T("Object Limit:  %lu\n"), appQuota.dwObjectLimit);
     _tprintf(_T("Object Size:   %lu\n"), appQuota.dwObjectSize);
     _tprintf(_T("Bytes Used:    %I64u\n"), appQuota.ulBytesUsed);
     _tprintf(_T("Bytes Free:    %I64u\n"), appQuota.ulBytesFree);
     _tprintf(_T("Storage Limit: %I64u\n"), appQuota.ulStorageLimit);
}
else
{
    // Unable to get the quota information for this account
    CString strError;

    appStorage.GetErrorString(strError);
    _tprintf(_T("Unable to get the account quota (%s)\n"), (LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetObjectInformation, GetObjectSize, WEB_STORAGE_QUOTA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The default timeout period is 10 seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

OpenStorage, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::GetTransferStatus Method

```
    LPWEB_STORAGE_TRANSFER lpStatus
);
```

The **GetTransferStatus** method returns information about the current data transfer in progress.

Parameters

lpStatus

A pointer to a WEB_STORAGE_TRANSFER structure which contains information about the status of the current data transfer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetTransferStatus** method returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this method will return the status of the last successful data transfer.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the storage session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetData, PutData, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can potentially allow the application to be re-entered, it is possible that another blocking method may be called while it is in progress. Because only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the storage client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

If your application attempts to perform multiple operations simultaneously, such as downloading multiple stored objects to the local system, each client session should be isolated in its own thread. This will avoid potential situations where a request is refused because a blocking data transfer is already in progress on the current thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetFile, GetData, IsConnected, IsInitialized, PutFile, PutData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if a connection has been established with the storage server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero. To get extended error information, call the **GetLastError** method.

Remarks

The client does not maintain a continuous, persistent connection with the storage server. The connection may be closed and reopened internally as needed. If the client session has been idle for a period of time, this method can return zero.

If the **GetLastError** method returns ST_ERROR_NOT_CONNECTED it means the client session is valid, however it not currently connected to the storage server. The next call to store or retrieve an object will the cause the client to reconnect automatically.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

IsBlocking, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on a different system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

CWebStorage, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::MoveObject Method

```
BOOL MoveObject(
   LPCTSTR lpszOldLabel,
   LPCTSTR lpszNewLabel,
   DWORD dwStorageType,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **MoveObject** method moves an existing object to a different storage container.

Parameters

lpszOldLabel

A null-terminated string which specifies the name of the existing storage object to be moved. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A null-terminated string which specifies a new label for the storage object being moved. This parameter may be NULL or point to an empty string, in which case the label name is not changed.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the object is renamed and not moved to another container. One of the following values may be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the new storage object. If this information is not required, this parameter may be omitted or a NULL pointer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **MoveObject** method is used to move an existing storage object to a new container. For example, it can move an object originally created using WEB_STORAGE_USER to the WEB_STORAGE_MACHINE container. The *dwStorageType* parameter must specify a valid storage container type and cannot be WEB_STORAGE_DEFAULT.

If the *dwStorageType* parameter specifies the same container that the object is currently in, and the *lpszNewLabel* parameter specifies a new label name, this method will simply rename the existing object and is effectively the same as calling the **RenameObject** method.

To duplicate an existing storage object, use the **CopyObject** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CopyObject, DeleteObject, RenameObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::OpenStorage Method

```
BOOL OpenStorage(
   DWORD dwStorageType
);

BOOL OpenStorage(
   LPCTSTR lpszAppId,
   DWORD dwStorageType
);
```

The **OpenStorage** method establishes a connection with the server and opens the storage container associated with the specified storage type.

Parameters

lpszAppId

A null-terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is omitted, NULL or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the WEB_STORAGE_GLOBAL container will be opened. One of the following values may be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this

	storage type will not be available to the application.

Return Value

If the method succeeds, the return value is a handle to the storage container. If the method fails, the return value is INVALID_HANDLE. To get extended error information, call the **GetLastError** method.

Remarks

The **OpenStorage** method opens the specified storage container and requests an access token for the application. This is the first method that must be called prior to accessing any stored objects.

The application ID is a string that uniquely identifies the application requesting the access and must have been previously registered with the server by calling the **RegisterAppId** method. If the *lpszAppId* parameter is NULL or an empty string, the method will use a default internal ID that is allocated for each storage account. You can use this default ID if you wish to share data between all of the applications you create.

The storage type specifies the type of container that objects will be stored in. In most cases, we recommend using WEB_STORAGE_GLOBAL which means that stored objects will be accessible to all users of your application. However, you can limit access to the stored objects based on the local domain, local machine ID or the current user SID.

If you specify anything other than global storage, objects can be orphaned if the system configuration changes. For example, if WEB_STORAGE_MACHINE is specified the objects that are stored can only be accessed from that system. If the system is reconfigured (for example, the boot volume formatted and Windows is reinstalled) the unique identifier for that system will change and the previous objects that were stored by your application can no longer be accessed.

It is advisable is to store critical application data and configuration information using WEB_STORAGE_GLOBAL and use other non-global storage containers for configuration information that is unique to that system and/or user which is not critical and can be easily recreated.

If a storage container was previously opened, calling this method will automatically close the current container and open the new container specified by the *dwStorageType* parameter. This can change the internal handle value used to reference the container and the old handle will no longer be valid.

Example

```
CWebStorage appStorage;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };

// Register the application ID which identifies the application
if (!appStorage.RegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
if (!appStorage.OpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}
```

```
// Upload a local file to the storage server and return information
// about the stored object when it completes
if (appStorage.PutFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    CString strError;

    appStorage.GetLastError(strError);
    _tprintf(_T("Cannot store \"%s\" (%s)\n"), lpszLocalFile, (LPCTSTR)strError);
}

// Release the handle allocated for this storage session
appStorage.CloseStorage();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseStorage, GetData, GetFile, GetStorageId, PutData, PutFile, RegisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::PutData Method

```
BOOL PutData(
   LPCTSTR lpszObjectLabel,
   LPCVOID lpvBuffer,
   DWORD dwLength,
   LPWEB_STORAGE_OBJECT lpObject
);

BOOL PutData(
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszObjectText,
   INT cchObjectText,
   LPWEB_STORAGE_OBJECT lpObject
);

BOOL PutData(
   LPCTSTR lpszObjectLabel,
   LPCVOID lpvBuffer,
   DWORD dwLength,
   LPCTSTR lpszContentType,
   DWORD dwAttributes,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **PutData** method stores the contents of buffer to a container and returns information about the new object.

Parameters

lpszObjectLabel

A null-terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The method will fail if the label contains any illegal characters.

lpvBuffer

A buffer that contains the data to be stored. If this parameter is NULL, the ***dwLength*** parameter must have a value of zero and a storage object will be created which has no data associated with it.

dwLength

An unsigned integer value that specifies the number of bytes that will be copied from the ***lpvBuffer*** parameter and stored in the object. If the ***lpvBuffer*** parameter is NULL, this value must be zero or the method will fail.

lpszObjectText

A null-terminated string which is used with an alternate version of this method. The string will be stored as a text object and the ***cchObjectText*** parameter will specify the number of characters to be stored.

cchObjectText

An integer value that specifies the number of characters that will be copied from the ***lpszObjectText*** string and stored in the object. If the ***lpszObjectText*** parameter is NULL, this value must be zero or the method will fail. If this parameter is omitted, the length of the string will be calculated by counting the number of characters up to the terminating null.

lpszContentType

A null-terminated string that identifies the contents of the buffer being stored. If this parameter is omitted, a NULL pointer, or specifies a zero-length string, the method will attempt to automatically determine the content type based on the object label and the contents of the buffer.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. If this parameter is omitted, the default value WEB_OBJECT_NORMAL will be used. This value can be a combination of one or more of the following bit flags using a bitwise OR operation:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **EnumObjects** method. The object can only be accessed directly when specifying its label.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that was created or replaced. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **PutData** method uploads the contents of a buffer to the current storage container. The content type, which identifies the type of data stored in the object, and its attributes may be specified by the caller or default values may be used.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text files have a content type of **text/plain** while an XML-formatted text file would have a content type of **text/xml**. Files that contain unstructured binary data are typically identified as **application/octet-stream**.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this method will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

If **PutData** method is used to store the contents of a string as a text object, the string should be terminated with a null character and cannot contain embedded nulls. If the Unicode version of this function is called, the contents of the string will be normalized prior to being converted to UTF-8 using canonical composition, where decomposed characters are combined to create their canonical precomposed equivalent.

The size of the text object that is created can be different from the length of the string buffer passed to this method, depending on how the text has been encoded and stored. This will almost always be the case when the Unicode version of this function is called because the UTF-16 string will be converted and stored as UTF-8 encoded text. This encoding will typically increase the size of the stored object unless the string only contains ASCII characters.

The version of the **PutData** method which accepts a string should only be used to store textual data in a null terminated string, and should never be used to store binary data. If you wish to create or update an object which contains binary data, you should always use use a byte array.

If you want to upload the contents of a file, the **PutFile** method simplifies this process. The example code below demonstrates how the Windows API can be used to read from a local file and store the contents using **PutData**.

If you are storing a large amount of data and want your application to receive progress updates during the data transfer, use the **RegisterEvent** method and provide a pointer to a static callback function that will receive event notifications.

Example

```
HANDLE hFile = INVALID_HANDLE_VALUE;
LPBYTE lpContents = NULL;
DWORD dwLength = 0;
BOOL bFileRead = FALSE;
WEB_STORAGE_OBJECT webObject;

// Open a file on the local system and read the contents
// into a buffer that will be stored on the server
hFile = CreateFile(lpszLocalFile,
                   GENERIC_READ,
                   0,
                   NULL,
                   OPEN_EXISTING,
                   FILE_ATTRIBUTE_NORMAL,
                   NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
    // Unable to open the file
    return;
}

// Get the size of the file and allocate a buffer large
// enough to store the contents of the file
dwLength = GetFileSize(hFile, NULL);
lpContents = (LPBYTE)LocalAlloc(LPTR, dwLength + 1);

if (lpContents == NULL)
{
    // Memory allocation failed
    return;
}

bFileRead = ReadFile(hFile, lpContents, dwLength, &dwLength, NULL);
CloseHandle(hFile);

if (!bFileRead)
{
```

```
    // Unable to read the contents of the file
    return;
}

// Store the contents of the buffer, identifying it as an unstructured
// stream of bytes, and the object is created with read/write access
if (pStorage->PutData(lpszObjectLabel, lpContents, dwLength, &webObject))
{
    // The object was created or replaced, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be created
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to create \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFile, GetData, PutFile, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::PutFile Method

```
BOOL PutFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject

BOOL PutFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszContentType,
   DWORD dwAttributes,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **PutFile** method uploads the contents of a local file to a storage container and returns information about the new object.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be uploaded. If a path is not specified, the file will be read from the current working directory. The current user must have read access to the file, and an error will be returned if the method cannot obtain an exclusive lock on the file during the upload process.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The method will fail if the label contains any illegal characters.

lpszContentType

A null-terminated string that identifies the contents of the file being uploaded. If this parameter is omitted, a NULL pointer, or specifies a zero-length string, the method will attempt to automatically determine the content type based on the file name extension and the contents of the file.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. If this parameter is omitted, a default value of WEB_OBJECT_NORMAL will be used. This value can be a combination of one or more of the following bit flags using a bitwise OR operation:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned

	when enumerated using the **EnumObjects** method. The object can only be accessed directly when specifying its label.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **PutFile** method uploads the contents of a local file to the current storage container. The content type, which identifies the type of data stored in the object, and its attributes may be specified by the caller or default values may be used.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text files have a content type of **text/plain** while an XML-formatted text file would have a content type of **text/xml**. Files that contain unstructured binary data are typically identified as **application/octet-stream**. If the content type is not explicitly specified, an attempt will be made to identify it automatically based on contents of the file and the file extension.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this method will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

The **ValidateLabel** method can be used to ensure a label is valid prior to calling this method. Refer to that method for more information about the difference between Windows file names and object labels

If you are uploading a large file and want your application to receive progress updates during the data transfer, use the **RegisterEvent** method and provide a pointer to a static callback function that will receive event notifications.

Example

```
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the storage container, automatically
// determining the content type with normal read/write access
if (pStorage->PutFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // The file was uploaded, display the object metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The file could not be uploaded, display the error
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to store \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetFile, RegisterEvent, UploadFile, ValidateLabel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::RegisterAppId Method

```
BOOL RegisterAppId(
   LPCTSTR lpszAppId
);
```

The **RegisterAppId** method registers a unique application identifier with the server.

Parameters

lpszAppId

A null-terminated string which identifies the application requesting access. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the method will fail. See the remarks below on the recommended method for identifying your application.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RegisterAppId** method registers an application ID with the server which uniquely identifies the application that is requesting access to the storage container. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

It is recommended that you use a standard format for the application ID that consists of your company name, application name and optionally a version number. For example:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

It is important to note that with these two example IDs, although they are similar, they reference two different applications. Objects stored using the first ID will not be accessible using the second ID. If you want to store objects that should be shared between all versions of the application, it is recommended that you use the first form, without the version number. If you want to store objects that should only be accessible to a specific version of your application, then it is recommended that you use the second form that includes the version number.

It is safe to call this method with an application ID that was previously registered. If the provided application ID has already been registered, this method will succeed. You can choose to call this method every time your application starts to ensure the application has been registered correctly.

If you no longer wish to use an application ID you have previously registered, you can call the **UnregisterAppId** method. Exercise caution when unregistering an application. This will cause all objects stored using that ID to be deleted by the storage server. Once an application ID has been unregistered, the operation is permanent. Calling **UnregisterAppId** and then **RegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

The application ID is intended to be an application defined human-readable string that uniquely identifies your application. If you want to obtain the internal storage ID associated with your

application, use the **GetStorageId** method. The storage ID is a fixed-length string of letters and numbers guaranteed to be unique across all applications that you register.

It is not required for your application to create a unique application ID. Each storage account has a default internal application ID named `SocketTools.Storage.Default`. This default ID is used if a NULL pointer or an empty string is specified to methods like **OpenStorage**. It is intended to identify storage available to all applications that you create.

To enumerate the application IDs registered with your storage account, use the **GetFirstApplication** and **GetNextApplication** methods.

Example

```
 CWebStorage appStorage;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };

// Register the application ID which identifies the application
if (!appStorage.RegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
if (!appStorage.OpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}

// Upload a local file to the storage server and return information
// about the stored object when it completes
if (appStorage.PutFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %I64u\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    CString strError;

    appStorage.GetLastError(strError);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}

// Release the handle allocated for this storage session
appStorage.CloseStorage();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CloseStorage, GetFirstApplication, GetNextApplication, GetStorageId, OpenStorage, UnregisterAppId, ValidateAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::RegisterEvent Method

```
BOOL RegisterEvent(
   UINT nEventId,
   WEBEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. This parameter cannot be zero.

Value	Description
WEB_EVENT_CONNECT	The connection to the storage server has been established and the session has been authenticated. This is the first event that occurs when initiating an operation to create or retrieve a storage object.
WEB_EVENT_DISCONNECT	The connection to the storage server has been closed and the session is terminating. This is the last event that occurs after completing an operation to create or retrieve a storage object.
WEB_EVENT_READ	The contents of an object is being read from the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of retrieving the object contents from the container.
WEB_EVENT_WRITE	The contents of an object is being written to the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of submitting the object contents to the container.
WEB_EVENT_TIMEOUT	The operation has exceeded the specified timeout period. The application may attempt to retry the operation or report an error to the user. This event typically indicates a connectivity problem with the storage server.
WEB_EVENT_CANCEL	The operation has been canceled. This event occurs after the application calls the **Cancel** method while an object is being stored or retrieved.
WEB_EVENT_PROGRESS	A storage operation is in progress. This event periodically occurs as the contents of a storage object is being read or written from the container. To retrieve information about the status of the operation, the

	application should register a handler for this event and call the **GetTransferStatus** method from within that handler.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback method, see the description of the **WebEventProc** callback method. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback method specified by ***lpEventProc***. If the application targets x86 (32-bit) platforms, this parameter must be a 32-bit unsigned integer. If the application targets x64 (64-bit) platforms, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RegisterEvent** method associates a static callback function with a specific event. The event handler is a **WebEventProc** function that is invoked when the event occurs. Arguments are passed to the method to identify the storage handle, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

Events are only generated as the result of a call to the **GetFile**, **GetData**, **PutFile** and **PutData** methods. Other storage methods will not generate event notifications.

This method is typically used to register an event handler that is invoked while the contents of a storage object is being transferred. The WEB_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one WEB_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The ***dwParam*** parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetTransferStatus, UnregisterEvent, WebEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::RenameObject Method

```
BOOL RenameObject(
   LPCTSTR lpszOldLabel,
   LPCTSTR lpszNewLabel
);
```

The **RenameObject** method changes the label associated with the storage object.

Parameters

lpszOldLabel

A null-terminated string which specifies the name of the existing storage object to be renamed. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A null-terminated string which specifies a new label for the storage object being moved. This parameter may not be NULL or point to an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RenameObject** method is used to change the label for an existing storage object. Although storage object labels are similar to Windows file names, they are case-sensitive. When renaming an object, your application must specify the original label name exactly as it was created. The object label cannot contain wildcard characters.

To duplicate an existing storage object, use the **CopyObject** method. If you need to move the object to a different storage container, use the **MoveObject** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CopyObject, DeleteObject, MoveObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::ResetStorage Method

```
BOOL ResetStorage();
```

The **ResetStorage** method resets the application storage container and deletes all stored objects.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The storage container contains information for each of the objects that have been stored using the handle returned by the **OpenStorage** method. Each of these objects are associated with both the application ID and the storage type that was specified by the caller. This method instructs the server to reset the container back to its initial state, deleting all of the objects that were stored in it.

Exercise caution when using this method. The reset operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DeleteObject, OpenStorage, RegisterAppId, UnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Method failure is typically indicated by a return value such as FALSE, NULL, INVALID_HANDLE or WEBAPI_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. An application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::SetTimeout Method

```
DWORD SetTimeout(
   DWORD dwTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

hClient

Handle to the client session.

dwTimeout

The number of seconds to wait for a storage operation to complete.

Return Value

If the method succeeds, the return value is the previous timeout period for the session. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The default timeout period is 10 seconds. The minimum timeout period is 5 seconds, and the maximum timeout period is 60 seconds. Values outside of this range will be normalized internally and will not cause the method to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

GetTimeout, OpenStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** method. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::UnregisterAppId Method

```
BOOL UnregisterAppId(
   LPCTSTR lpszAppId
);
```

The **UnregisterAppId** method unregisters the application identifier and deletes all associated storage objects.

Parameters

lpszAppId

A null-terminated string which specifies the application ID to be deleted. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the method will fail.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **UnregisterAppId** method deletes the internal storage identifier associated with the application ID and revokes all access tokens that were granted for the application. This operation is immediate and permanent.

Exercise caution when using this method. This will permanently delete all objects that were stored for the specified application. Calling **UnregisterAppId** and then **RegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

This method cannot be used to unregister the default storage application ID `SocketTools.Storage.Default`. If this ID is specified, the method will fail with an error indicating that the ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

OpenStorage, RegisterAppId, ResetStorage, ValidateAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::UnregisterEvent Method

```
BOOL RegisterEvent(
   UINT nEventId
);
```

The **UnregisterEvent** method removes an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. This parameter cannot be zero. The following values may be used:

Value	Description
WEB_EVENT_CONNECT	The connection to the storage server has been established and the session has been authenticated. This is the first event that occurs when initiating an operation to create or retrieve a storage object.
WEB_EVENT_DISCONNECT	The connection to the storage server has been closed and the session is terminating. This is the last event that occurs after completing an operation to create or retrieve a storage object.
WEB_EVENT_READ	The contents of an object is being read from the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of retrieving the object contents from the container.
WEB_EVENT_WRITE	The contents of an object is being written to the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of submitting the object contents to the container.
WEB_EVENT_TIMEOUT	The operation has exceeded the specified timeout period. The application may attempt to retry the operation or report an error to the user. This event typically indicates a connectivity problem with the storage server.
WEB_EVENT_CANCEL	The operation has been canceled. This event occurs after the application calls the **Cancel** method while an object is being stored or retrieved.
WEB_EVENT_PROGRESS	A storage operation is in progress. This event periodically occurs as the contents of a storage object is being read or written from the container. To retrieve information about the status of the operation, the application should register a handler for this event and call the **GetTransferStatus** method from within that handler.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **UnregisterEvent** method removes the association between a specific event and its callback function. This method can be used if the application no longer requires the event notification. All event handlers for the client session are automatically removed when the **CloseStorage** method is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetTransferStatus, RegisterEvent, WebEventProc,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::UploadFile Method

```
BOOL UploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject
);

BOOL UploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszAppId,
   DWORD dwStorageType,
   DWORD dwAttributes,
   DWORD dwTimeout,
   LPWEB_STORAGE_OBJECT lpObject,
   WEBEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **UploadFile** method uploads the contents of a local file and creates or overwrites a storage object.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A null-terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is omitted, NULL or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the default value of WEB_STORAGE_GLOBAL will be used. One of the following values may be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as

	defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. If this parameter is omitted, a default value of WEB_OBJECT_NORMAL will be used. This value can be a combination of one or more of the following bit flags using a bitwise OR operation:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **CWebStorage::EnumObjects** method. The object can only be accessed directly when specifying its label.

dwTimeout

An unsigned integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback method, see the description of the **WebEventProc** callback function. If this parameter is NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback function specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **UploadFile** method uploads the contents of a local file and either creates a new storage object, or replaces an object if one already exists with the same label. The **UploadFile** method provides a simpler interface that defaults to uploading an object to the global storage container.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this method will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are uploading a large file and want your application to receive progress updates during the data transfer, provide a pointer to a static callback function as the *lpEventProc* parameter. That function will receive event notifications as the data is being uploaded.

Example

```
CWebStorage appStorage;
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the global storage container
if (appStorage.UploadFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // The object was uploaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be uploaded, display the error
    CString strError;

    appStorage.GetLastError(strError);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DownloadFile, GetFile, PutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::ValidateAppId Method

```
BOOL ValidateAppId(
   LPCTSTR lpszAppId
);
```

The **ValidateAppId** method validates the specified application identifier.

Parameters

lpszAppId

A null-terminated string which specifies the application ID to be validated. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the method will fail.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ValidateAppId** method is used to determine if the specified application identifier is valid and has been previously registered using the **RegisterAppId** method. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RegisterAppId, UnregisterAppId, ValidateLabel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage::ValidateLabel Method

```
BOOL ValidateLabel(
   LPCTSTR lpszObjectLabel
);
```

The **ValidateLabel** method validates the specified object label.

Parameters

lpszObjectLabel

A null-terminated string which specifies the object label to be validated. This parameter cannot be NULL or point to a zero-length string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

Object labels are similar to Windows file names, except they are case-sensitive. The maximum length of a label string is 512 characters, including the terminating null character. Leading and trailing whitespace (spaces, tabs, linebreaks, etc.) are ignored in label names.

Illegal characters include ASCII and Unicode control characters 1 through 31, single quotes (39), double quotes (34), less than symbol (60), greater than symbol (62), pipe (124), asterisk (42) and question mark (63). A null character (0) specifies the end of the label and any subsequent characters are ignored. It is not possible to embed null characters in the label name.

Label names may contain forward slash (47) characters and backslash (92) characters, however it is important to note that objects are not stored in a hierarchical structure. An application can create its own folder-like structure to the labels it creates, but this structure is not imposed or enforced by the library.

If the application is built to use Unicode, labels can contain Unicode characters which are internally encoded as UTF-8. This is important to consider if you have an project built using a multi-byte (ANSI) character set and it needs to access an object that was created using Unicode characters. In that case, the ANSI application must be prepared to handle UTF-8 encoded names and display them appropriately.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RegisterAppId, UnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebStorage Data Structures

- SYSTEMTIME
- WEB_STORAGE_APPLICATION
- WEB_STORAGE_QUOTA
- WEB_STORAGE_OBJECT
- WEB_STORAGE_TRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_APPLICATION

The **WEB_STORAGE_APPLICATION** structure contains information about a registered application.

```
typedef struct _WEB_STORAGE_APPLICATION
{
    TCHAR      szAppId[64];
    TCHAR      szApiKey[64];
    TCHAR      szLuid[64];
    DWORD      dwTokens;
    DWORD      dwReserved;
    SYSTEMTIME stCreated;
    SYSTEMTIME stUpdated;
} WEB_STORAGE_APPLICATION, *LPWEB_STORAGE_APPLICATION;
```

Members

szAppId

A null-terminated string which contains the application ID that was registered using the **RegisterAppId** method.

szApiKey

A null-terminated string which contains the API key value that is used internally by the storage services. The AppID is effectively the human readable alias for this key value used to identify the stored objects for the application. This value is guaranteed to be unique across all applications registered with the storage service.

szLuid

A null-terminated string which specifies a locally unique value associated with the registered application. This value is used internally by the storage service and guaranteed to be unique to the storage account that has registered the application. Note that there are no functions that currently accept the application LUID as parameter, but you may choose to use this value in your own application for other purposes.

dwTokens

An unsigned integer value which specifies the number of access tokens associated with the registered application. Typically there is only one access token associated with a given AppID at any one time, although it is possible that the API may allocate additional access tokens under some circumstances.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

stCreated

A SYSTEMTIME structure that specifies the date and time the AppID was registered. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stUpdated

A SYSTEMTIME structure that specifies the date and time the AppID object was last updated. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

This structure is used in conjunction with the **GetFirstApplication** and **GetNextApplication** methods which enumerate all registered applications for the current storage account.

To adjust the creation and update times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** function. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_OBJECT

The **WEB_STORAGE_OBJECT** structure contains information about an individual storage object.

```
typedef struct _WEB_STORAGE_OBJECT
{
    TCHAR      szObjectId[64];
    TCHAR      szLabel[512];
    TCHAR      szDigest[128];
    TCHAR      szContent[128];
    DWORD      dwAttributes;
    DWORD      dwObjectSize;
    SYSTEMTIME stCreated;
    SYSTEMTIME stModified;
} WEB_STORAGE_OBJECT, *WEB_STORAGE_OBJECT;
```

Members

szObjectId

A null-terminated string which contains the unique identifier associated with this object. Object IDs are guaranteed to be unique for each storage object that is created by the application. The maximum length of an object ID is 64 characters, including the terminating null character.

szLabel

A null-terminated string which contains the label assigned to the object by the application. Object labels are case-sensitive and must be unique for each object. An application uses labels to reference an object with a human-recognizable name, rather than referencing them by their object ID. The maximum length of an object label is 512 characters, including the terminating null character.

szDigest

A null-terminated string which specifies the digest of the object contents, computed using an SHA-256 hash. The maximum length of the ***szDigest*** string is 128 characters, including the terminating null character. However, the digest value is always represented as a string of hexadecimal numbers that is exactly 64 characters long. It is important to note that even a zero-length object will have a digest, which is the standard SHA-256 NULL hash value.

szContent

A null-terminated string which specifies the MIME content type for the storage object. The content type is determined by the object label and evaluating the contents of the object. It is also possible for the application to explicitly specify the content type of the object when it is created.

dwAttributes

An unsigned integer value that specifies the attributes for the storage object. The object attributes are comprised of one or more bit flags:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application.

	Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **EnumObjects** method. The object can only be accessed directly when specifying its label.

dwObjectSize

An unsigned integer value that specifies the size of the storage object in bytes. The maximum size of an individual object is determined by the storage quota limits established for the account.

stCreated

A SYSTEMTIME structure that specifies the date and time the storage object was created. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stModified

A SYSTEMTIME structure that specifies the date and time the storage object was last modified. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

The object content type will always be in the format **type/subtype** where the type specifies a common media type (e.g.: text, audio, video, etc.) and subtype specifies the specific content. The most common content type for text files is **text/plain**. If the content type is unknown, the default content type is **application/octet-stream**.

Text objects may also optionally include the character encoding as part of the content type. For example, if an object contains UTF-8 encoded text, the content type may be returned as **text/plain; charset=utf-8**. If your application is parsing the content types, you must check if a character encoding was also included in the value. Text objects that do not specify an encoding either contain ASCII or text which uses the system code page. Unicode text will always be stored using UTF-8 encoding.

To adjust the object creation and modification times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** method. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_QUOTA

The **WEB_STORAGE_QUOTA** structure contains information about the current storage account data usage and limits.

```
typedef struct _WEB_STORAGE_QUOTA
{
    DWORD     dwObjects;
    DWORD     dwObjectLimit;
    DWORD     dwObjectSize;
    ULONGLONG ulBytesUsed;
    ULONGLONG ulBytesFree;
    ULONGLONG ulStorageLimit;
} WEB_STORAGE_QUOTA, *LPWEB_STORAGE_QUOTA;
```

Members

dwObjects

An unsigned integer value which specifies the number of storage objects allocated for the account. This value may not exceed the total number of objects specified by the ***dwObjectLimit*** member.

dwObjectLimit

An unsigned integer value which specifies the maximum number of storage objects that may be created. In addition to the limit on the total amount of storage that may be used, there is a limit on the total number of objects that may be created by all applications.

dwObjectSize

An unsigned integer value which specifies the maximum size of an individual storage object. In addition to a limit on the total amount of storage used and the number of objects created, each object stored by the application cannot exceed this size.

ulBytesUsed

An unsigned 64-bit integer value which specifies the total number of bytes of data allocated for all storage objects. This value may not exceed the total number of bytes of storage available, which is specified by the ***ulStorageLimit*** member.

ulBytesFree

An unsigned 64-bit integer value which specifies the number of bytes available for the storage of new objects. This value reflects the total amount of available storage across all applications registered with the development account. If this value is zero, your storage account has reached its storage limit.

ulStorageLimit

An unsigned 64-bit integer value which specifies the maximum number of bytes of data storage available. This limit applies to all applications registered with the development account.

Remarks

Storage quota limits are assigned for each SocketTools development account. The **GetStorageQuota** method will populate this structure with information about the limits on your account. Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

These values do not represent limits on storage usage by a specific application. Quotas limits apply to all applications that are registered with the development account, which is identified with the runtime

license key used to initialize the class instance.

If your storage quota has been exceeded, either because the total number of objects or the total bytes of storage has reached their limit, your applications will be unable to create new objects. Your application can continue to access existing objects, regardless of your current quota limits.

To free storage space, use the **DeleteObject** method to delete individual storage objects that are no longer needed by your application, or use the **ResetStorage** method to delete all objects in a container.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

DeleteObject, GetStorageQuota, ResetStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_TRANSFER

This structure is used by the **CWebStorage::GetTransferStatus** method to return information about a data transfer in progress.

```
typedef struct _WEB_STORAGE_TRANSFER
{
   TCHAR     szObjectLabel[512];
   DWORD     dwStorageType;
   DWORD     dwBytesTotal;
   DWORD     dwBytesCopied;
   DWORD     dwBytesPerSecond;
   DWORD     dwTimeElapsed;
   DWORD     dwTimeEstimated;
} WEB_STORAGE_TRANSFER, *LPWEB_STORAGE_TRANSFER
```

Members

szObjectLabel

A null-terminated string that specifies the label for the object that is being retrieved, created or replaced.

dwStorageType

An integer value that identifies the storage container type. This will be one of the following values:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwBytesTotal

An unsigned integer which specifies the total number of bytes that will be transferred. If the object is being downloaded to the local host, this is the size of the stored object. If the data is being uploaded from the local host to be stored on the server, it is the size of the buffer or local file.

dwBytesCopied

An unsigned integer which specifies the total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Class

The SocketTools WebView class provides an embedded Chromium-based web browser that can be integrated into desktop applications. The class is implemented using the Microsoft Edge WebView2 runtime and provides a much simpler API which can be used with most languages, without requiring direct use of the COM-based interfaces in the WebView2 SDK.

Reference

- Methods
- Constants
- Data Structures
- Error Codes

Library Information

File Name	CSHTMV12.DLL
Version	12.0.3024.1286
LibID	175ACE49-FE5D-46C8-A0EF-5188C97B993D
Import Library	CSHTMV12.LIB
Dependencies	None

Overview

The SocketTools CWebView class makes it possible to embed modern web content into Windows applications using a simple procedural interface. Developers can work entirely with standard data types and function calls, avoiding the complexity of COM programming or .NET interop layers. The API is designed to be language-agnostic and can be called from C or C++, as well as other environments that support invoking functions from a dynamic link library such as Object Pascal and Visual Basic. Applications can use a traditional Windows GUI or be console-based, and functions are thread-safe where appropriate.

Because the class encapsulates the creation of the WebView2 environment and its associated controller, applications require only minimal setup. Developers can host a browser window, execute scripts, or capture output with just a few lines of code. Advanced features such as asynchronous execution, response header enumeration, and event callbacks are supported, but optional, making it easy to start simple and grow into more complex scenarios.

Applications are not limited to navigating to external websites. You can dynamically generate HTML content within your program and load it directly into the browser, making it possible to render reports, status pages, or other application data without creating temporary files. The browser also supports displaying local content using `file://` URLs, including HTML, images, PDF documents, and other formats that WebView2 can natively render. This makes it easy to blend remote and local resources within the same user interface.

When connecting to web servers, the control uses the underlying WebView2 runtime and supports modern versions of HTTP, including HTTP/1.1, HTTP/2, and HTTP/3 where available. Connections are secured with Transport Layer Security (TLS), with support for TLS 1.2 and TLS 1.3 by default, and certificate validation is performed automatically by the runtime. This ensures that applications embedding the browser benefit from the same security standards as Microsoft Edge, including

enforcement of current best practices for encryption and certificate handling.

Unlike most SocketTools libraries which return pseudo-handles to reference client sessions, the WebView class uses window handles (HWND) for nearly all of its functions. The only exceptions are the functions used to create or attach to a WebView instance. This design provides a direct connection between the class and the underlying windowing system, and ensures consistent behavior with standard Windows APIs. Even in console applications, the class creates a host window which provides the handle used with the other functions. Using window handles also makes integration with GUI frameworks such as MFC, WinForms, or WPF more natural, since those frameworks are already designed around standard Windows window management.

Requirements

It is not necessary to download the Microsoft.Web.WebView2 SDK NuGet package and include it in your project to use this class. However, the end-user system must have the current version of the Edge WebView2 Runtime installed. On Windows 11 and Windows 10 version 2004 (build 19041) and later, the runtime is considered part of the operating system and is updated automatically. On earlier platforms such as Windows 7, Windows 8, and Windows 10 builds prior to 19041, you may need to redistribute or install the WebView2 runtime separately. The class includes functions which can tell you if the runtime is installed and what version is available.

For Windows 7 and Windows 8, the last version of the WebView2 runtime available is 109.0.1518.140. This version is supported by SocketTools, however it is important to note that Microsoft is no longer updating the control on those platforms. This means it will no longer receive security updates or other improvements; some features may work differently on older platforms, and newer features will not be available.

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this class, it is recommended that you install the library in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

In addition to the SocketTools library, the WebView2 Runtime must be present on the target system. On current versions of Windows 10 and Windows 11 the runtime is treated as part of the operating system and is updated automatically. On earlier platforms, you may need to redistribute or install the runtime using Microsoft's evergreen installer. We recommend always using the evergreen runtime rather than the fixed version, as it ensures security updates and feature improvements are delivered through Windows Update without requiring changes to your application.

The WebView2 Runtime is available as a separate download from Microsoft. For current links and redistribution guidelines, please refer to the SocketTools knowledge base or Microsoft's official documentation.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Class Methods

Class	Description
CWebView	Constructor which initializes the current instance of the class.
~CWebView	Destructor which releases resources allocated by the class.
Method	Description
AsyncExecute	Execute JavaScript asynchronously in the current document.
Attach	Attach the browser to an existing host window.
CancelDownload	Cancel the specified file download.
CancelNavigation	Cancel the current navigation and stop loading.
ClearCache	Clear the cache of viewed pages for the current user profile.
ClearCookies	Delete cookies for the current user profile or session.
ClearHeaders	Clear the request and/or response headers for the current session.
ClearHistory	Clear the navigation history for the current session.
Close	Close the window and release all browser resources.
Create	Create an instance of the browser in a new window.
Detach	Detach the browser from its host window without destroying the host window.
DisableEvents	Disable event handling for the specified browser.
EnableEvents	Enable event handling for the specified browser.
EnumDownloads	Enumerate the file identifiers for all downloads during the current session.
Execute	Execute JavaScript and return the result synchronously.
FindBrowser	Locate the associated host or child window handle for an instance.
FreezeEvents	Suspend or re-enable event notifications for the specified browser.
GetAllOptions	Return the current option flags for the browser instance.
GetBackgroundColor	Get the background color used behind the page content.
GetBrowserVersion	Get the installed browser runtime version information.
GetChildWindow	Get the handle of the internal browser child window.
GetContentType	Return the Content-Type from the last response, if available.
GetCurrentUrl	Get the URL of the current document.
GetDocumentLength	Return the length of the current document.
GetDocumentText	Return the content for of the current document.
GetDocumentTitle	Return the document title.
GetDownloadInfo	Return information about the status of the specified download.
GetDownloadPath	Return the full path for the downloaded file.
GetErrorString	Return a description for the specified error code.

GetFilteredText	Return the current document text without HTML formatting.
GetFirstHeader	Begin enumerating response headers and returns the first name/value pair.
GetFocus	Return whether the browser control currently has input focus.
GetHeader	Get the value of a specific response header by name.
GetHeaderCount	Return the number of response headers from the last navigation.
GetHttpStatus	Return the HTTP status code from the last completed navigation.
GetLastError	Return the last error code.
GetNavigationStatus	Return the current navigation state.
GetNavigationTime	Return the elapsed time for the last navigation.
GetNextHeader	Continue header enumeration and return the next name/value pair.
GetOption	Checks whether a specific option flag is enabled.
GetSafeHwnd	Return the handle to the window hosting the browser control.
GetSecurityInfo	Return certificate and host security details for the last navigation.
GetSecurityLevel	Return the evaluated security level for the last navigation.
GetTimeout	Get the number of seconds to wait for a navigation to complete.
GetUserAgent	Return the current User-Agent value for this instance.
GetUserDataFolder	Return the full path to the current user data folder.
GetWebMessage	Retrieve the next pending message sent from the current document.
GetWebResult	Return the result from the most recent completed script execution.
GetZoomLevel	Return the current zoom level as a percentage.
GoBack	Navigate backwards through the browser history.
GoForward	Navigate forwards through the browser history.
Hide	Hide the browser window.
InEvent	Determine if the current thread is executing code within an event handler.
IsAttached	Determine if the class instance is attached to a browser window.
IsInitialized	Determine if the class instance has been initialized.
IsSecure	Determine if the current page was retrieved over a secure connection.
IsVisible	Determine if the browser window is currently visible.
KeyEventHandled	Set an accelerator key event as handled to suppress default processing.
Navigate	Navigate to the specified URL or local file path.
NormalizeUrl	Normalize a URL or path for safe, consistent navigation.
PostWebMessage	Post a string or JSON message to the current page.
PrintDocument	Display the browser print dialog for the current document.
Redraw	Force the host window and browser control to redraw themselves.
Reload	Reload the current document and optionally clear the cache.
Reset	Reset the embedded runtime control.

Resize	Resize the browser inside the host window.
RuntimeAvailable	Determine whether the required runtime is installed and available.
SaveDocument	Save the current document to a local file.
SetBackgroundColor	Set the background color used behind the page content.
SetBounds	Set the position and size of the browser child window.
SetDocumentText	Replace the current document with the specified HTML or plain text.
SetDownloadPath	Change the local file path for the specified download.
SetFocus	Give keyboard focus to the browser control.
SetHeader	Set or override a request header for subsequent navigations.
SetIcon	Set custom large and small icons for the host window.
SetLastError	Set the last error code.
SetOption	Enable or disable a specific option.
SetTimeout	Set the number of seconds to wait for a navigation to complete.
SetUserAgent	Set the default User-Agent string for this instance.
SetZoomLevel	Set the zoom level for the current view.
Show	Show the browser window.
WaitForEvent	Wait until a specific event occurs or the timeout period expires.
WaitUntilClosed	Wait until the host window is closed or destroyed.
WebViewEventProc	An application-defined callback function that processes events generated by the browser.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::CWebView

```
CWebView();
```

The **CWebView** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **Initialize** method to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

The calling thread must be in a single-threaded apartment (STA). During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), initialization will fail. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

Before creating an instance of this class, applications may optionally call **CWebView::RuntimeAvailable** to determine if the Microsoft Edge WebView2 runtime is installed and meets the minimum required version. This can be useful during application startup to provide a clear error message or guidance to the user if the runtime is missing or out of date, rather than attempting to initialize the control and handling the failure afterward.

On modern versions of Windows, the WebView2 runtime is included as part of the operating system and is serviced through Windows Update. However, on older or unsupported versions of Windows the runtime may not be present by default and must be installed separately. In these cases, your application needs to detect the missing runtime and prompt the user to install it.

The latest WebView2 runtime version officially supported on Windows 7 and Windows 8/8.1 is version 109.0.1518.140, released January 2023. After version 109, no further updates, new features, or fixes have been provided for those operating systems. Although this library supports Windows 7 and Windows 8, these platforms are not recommended because they no longer receive security updates for the browser.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

~CWebClient, Attach, Create, Detach, IsInitialized, RuntimeAvailable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::~CWebView

```
~CWebView();
```

The **CWebView** destructor releases resources allocated by the current instance of the **CWebView** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CWebView** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages. The WebView2 instance created by the class will be destroyed. If the class instance was created using a temporary user data folder, all cached data for the session will be deleted and the folder will be removed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

CWebViewClient, Attach, Create, Detach

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::AsyncExecute Method

```
BOOL AsyncExecute(
   LPCTSTR lpszJavaScript,
   DWORD_PTR dwParam
);
```

The **AsyncExecute** method executes a script on the current page, returning the results asynchronously.

Parameters

lpszJavaScript

A null-terminated string which specifies the JavaScript that should be executed in the context of the current page being viewed. This parameter is required and cannot be NULL.

dwParam

An optional parameter which can be used to provide context to the script being executed asynchronously. This user-defined value is passed back to the event handler which is invoked when the result from the script is returned back to the application. If this information is not required, this parameter can be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **AsyncExecute** method evaluates a JavaScript expression within the context of the currently loaded page and returns the result asynchronously. Unlike **Execute**, this method does not block while waiting for the script to complete. Instead, the result is delivered through the `WEBVIEW_EVENT_SCRIPT_COMPLETED` event. The application can use the dwParam value to correlate the completion event with the original request.

The script must be valid JavaScript and should not assume synchronous execution. Results are returned by calling **GetWebResult** which will provide you with the response in JSON-encoded form; for example, string values will be quoted. Because script execution is dispatched to the underlying WebView2 control, this method will fail if navigation is still in progress. Applications should always wait for the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event before invoking script execution.

Example

```
// A simple event handler implementation
VOID CALLBACK WebViewEventProc(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    CWebView *pWebView = reinterpret_cast<CWebView *>(dwParam);

    if (!pWebView)
        return;

    if (uEvent == WEBVIEW_EVENT_SCRIPT_COMPLETED)
    {
        BOOL bSuccess = (BOOL)wParam;

        if (bSuccess)
```

```
        {
            TCHAR szResult[512];
            if (pWebView->GetWebResult(szResult, _countof(szResult)))
            {
                // Get the response from the script
                // szResult will be a null-terminated string which may contain
                // a JSON object or JSON string (quoted and escaped if needed)
            }
        }
        else
        {
            // The script could not be executed
        }
    }
}

void RunExample(CWebView *pWebView)
{
    // Execute a simple script asynchronously
    if (!pWebView->AsyncExecute(_T("document.title"), reinterpret_cast<DWORD_PTR>
(pWebView)))
    {
        DWORD dwError = pWebView->GetLastError();
        // Handle the error condition
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Execute, GetWebResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Attach Method

```
BOOL Attach(
   HWND hWndHost,
   LPCTSTR lpszUrl
);

BOOL Attach(
   HWND hWndHost,
   DWORD dwOptions,
   LPCTSTR lpszUrl,
   LPCTSTR lpszLanguage,
   LPCTSTR lpszDataFolder
);
```

The **Attach** method attaches a new instance of the browser control to a host window.

Parameters

hWndHost

A handle to the window that will serve as a container for the browser control. This may either be a top-level application window or a child window. If the handle does not specify a valid window, the method will fail. This parameter is required and cannot be NULL.

dwOptions

One or more option bitflags. For more information, refer to the Options documentation. Most applications should not use a value of zero for this parameter. If no options are specified, the browser instance will be created with a very restrictive set of permissions that can prevent many websites from functioning correctly.

lpszUrl

A null-terminated string which specifies the initial URL the browser should navigate to after the control has been attached to the window. If this parameter is NULL or an empty string, the browser instance will be created with a blank page. The only schemes permitted are `https://`, `http://` and `file://`. If a domain name is specified without a scheme, it will default to using `https://`.

lpszLanguage

A null-terminated string which specifies an alternate language to use when the browser displays a context menu or dialogs. This value can be a standard language tag in BCP-47 format (such as "en-US") or it can be a string which specifies a hexadecimal Windows locale identifier (such as "0x0409"). It is recommended you use the language tag whenever possible, as Microsoft is deprecating the use of locale identifiers (LCIDs) and their associated APIs. If this parameter is a NULL pointer or an empty string, the browser will use the current default language for the local user.

lpszDataFolder

A folder name where the user profile data will be stored. This is the location where the browser will cache viewed pages, store cookies and form data, and where scripts may store session data. This value may be just a name (such as the name of the application) or an absolute path. If the folder name contains environment variables surrounded in percent (%) symbols, they will be replaced with their value in the current user's environment. If folder name does specify an absolute path, the profile will be stored in the current user's LocalAppData folder. If this parameter is NULL or an empty string, a temporary profile will be created for the session and then automatically deleted when the session ends.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Attach** method binds a new WebView instance to an existing top-level or child window you have already created. The host window must be created on the current UI thread and remain valid for the lifetime of the WebView. Do not call this method from a worker thread or for a window that was created by another thread; if a background thread needs a browser, create a dedicated window on that thread or use **Create** instead. Only one WebView instance can be attached to a given host window at a time. If you need to recreate the browser, call **Detach** before attaching again.

The calling thread must be in a single-threaded apartment (STA). During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), the attach will fail. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

The *dwOptions* parameter enables you to specify what features are enabled in the browser. It is recommended that most applications use `WEBVIEW_OPTION_ENABLE_DEFAULT` which enables functionality which is most commonly found in web browsers. If you want your host application to exchange messages with scripts executing in the browser, you should also specify the `WEBVIEW_OPTION_ENABLE_MESSAGES` options. This option must be enabled to use **PostWebMessage** and **GetWebMessage** methods.

When clicking on a link would normally open that page in a new tab or browser window, it will be redirected to the primary browser window by default. This ensures that the browser control retains focus and all events are within the context of that window. If you wish to permit the creation of popup windows and/or support the creation of new browser windows (for example, if a link target is "_blank") then specify the `WEBVIEW_OPTION_ENABLE_NEW_WINDOWS` option. Note that if you permit the creation of new popup windows, it can shift focus away from your main browser window. There is no mechanism in the API to enumerate the child popup windows and they are independent of the window attached to the browser.

By default, the browser will use the language associated with the current user's locale. If you specify an alternate language, this also affects what languages the browser includes with the **Accept-Language** request header. This will provide hints to the website as to what language is preferred when it returns content to the browser. However, it does not guarantee that a website will return content in that language and will not automatically cause a page to be translated from one language to another.

If you pass NULL or an empty string for the *lpszDataFolder* parameter, the library creates a temporary, per-session user-data folder under the current user's `LocalAppData` folder. That folder is flagged as temporary and is automatically deleted when you detach the WebView from the window via **Detach**. If you pass a relative path (such as "MyApp"), the library creates or uses that folder beneath `%LocalAppData%\WebView`, creating a folder path like `%LocalAppData%\WebView\MyApp`.

If you pass a fully qualified path as the *lpszDataFolder* parameter, it will use that path. However, it will not accept paths to unsafe locations and will return an error if your application does not have permission to create a folder in that location. If there are environment variables surrounded by percent (%) symbols, those variables will be expanded when the folder path is resolved. If you enable privacy mode with `WEBVIEW_OPTION_ENABLE_PRIVACY`, the library will ensure the profile is unique to the session by appending a GUID to the base folder and treating it as temporary. The **GetUserDataFolder** method can be used to obtain the full path to the data folder being used.

When multiple WebView instances point at the same user-data folder, they share a single WebView2

profile created with that folder path. In this configuration the browser cache, cookies, history, and site storage are persisted under that folder and are reused across windows and processes, which speeds up subsequent navigations and preserves sign-in/session state. The library passes your folder directly to WebView2 when creating the environment, so any instance using that same path participates in the same profile, and you can clear portions of that profile at runtime via the provided helpers for cache, history, and cookies.

Your **WEBVIEWEVENTPROC** event callback function is invoked on the same thread that owns the WebView window. Because events are raised on the UI thread, long-running work in an event handler can block the user interface and stall navigation. Where possible, defer heavy operations to background threads. You should avoid displaying modal dialogs or message boxes from within event handlers. Also note that some events may fire multiple times during a single navigation. For example, the security state can transition through several levels (unknown, neutral, secure, or warning) before settling on a final value. Applications should be written to handle these changes incrementally.

To present correctly inside a resizable child window, resize the browser view when the host window size changes. The library tracks the host window position but relies on your WM_SIZE handling to drive layout.

Example

```
#include "cstools12.h"

// Windows message handler
static LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// WebView event handler for navigation and security events
static VOID CALLBACK WebViewEvents(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    switch (uEvent)
    {
    case WEBVIEW_EVENT_NAVIGATION_STARTING:
        OutputDebugString(_T("[WebView] Navigation starting\n"));
        break;

    case WEBVIEW_EVENT_NAVIGATION_COMPLETED:
        if (wParam)
            OutputDebugString(_T("[WebView] Navigation completed\n"));
        else
            OutputDebugString(_T("[WebView] Navigation failed\n"));
        break;

    case WEBVIEW_EVENT_SECURITY_CHANGED:
        OutputDebugString(_T("[WebView] Security changed\n"));
        break;
    }
}

INT WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR
lpCmdLine, INT nCmdShow)
{
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        MessageBox(NULL, _T("Unable to initialize the WebView API"), _T("Error"),
MB_ICONEXCLAMATION);
        return 0;
    }
```

```
    LPCTSTR kClass = _T("MyWebViewWndClass");
    WNDCLASS wc{};
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszClassName = kClass;
    RegisterClass(&wc);

    HWND hWnd = Create(0, kClass, _T("WebView"),
                               WS_OVERLAPPEDWINDOW,
                               CW_USEDEFAULT, CW_USEDEFAULT, 1024, 720,
                               NULL, NULL, hInstance, NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // A standard Windows message loop
    MSG uMsg;

    while (GetWebMessage(&uMsg, nullptr, 0, 0))
    {
        TranslateMessage(&uMsg);
        DispatchMessage(&uMsg);
    }

    WebViewUninitialize();
    return (int)uMsg.wParam;
}

static LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
        // Create a WebView2 browser instance and attach to this window
        {
            DWORD dwOptions = WEBVIEW_OPTION_ENABLE_DEFAULT;
            LPCTSTR pszInitialUrl = _T("https://example.com");
            LPCTSTR pszDataFolder = _T("MyTestApp");

            if (!WebViewAttach(hWnd, dwOptions, pszInitialUrl, NULL, pszDataFolder,
WebViewEvents, 0))
            {
                 MessageBox(NULL, _T("Unable to attach the WebView2 control"),
_T("Error"), MB_ICONEXCLAMATION);
            }
        }
        return 0;

    case WM_SIZE:
        // Make sure the browser window resizes with the window
        {
            RECT rc;
            GetClientRect(hWnd, &rc);
            WebViewSetBounds(hWnd, rc.left, rc.top, rc.right - rc.left, rc.bottom -
rc.top);
```

```
        }
        return 0;

    case WM_DESTROY:
        // Detach the WebView2 browser instance from the window
        WebViewDetachWindow(hWnd);
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Attach, Create, DetachWindow, GetUserDataFolder, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::CancelDownload Method

```
BOOL CancelDownload(
   DWORD dwFileId
);
```

The **CancelDownload** method cancels the specified download in progress.

Parameters

dwFileId

An integer which specifies a unique file identifier. This value cannot be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CancelDownload** method causes the specified file download to be canceled. If the transfer has completed or has already been canceled, this method has no effect. This method does not wait for the download to be canceled before it returns; it returns immediately after asynchronously requesting the download be canceled. The **WEBVIEW_EVENT_DOWNLOAD_COMPLETED** event will fire once the download has stopped.

The *dwFileId* value is obtained when the **WEBVIEW_EVENT_DOWNLOAD_STARTING** event is fired and the *lParam* event argument will contain the unique file identifier for the download. You should treat the identifier as an opaque integer and not depend on the value being sequential. It is only guaranteed that the value will be unique for all downloads during the session and is only valid for the browser that initiated the download. Note there can be multiple downloads in progress simultaneously.

The **GetDownloadInfo** method can be used to get the status of the download and determine if it completed successfully, failed or was canceled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

CancelNavigation, EnumDownloads, GetDownloadInfo, GetDownloadPath, SetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::CancelNavigation Method

```
BOOL CancelNavigation();
```

The **CancelNavigation** method cancels any current or pending navigations.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method can be safely called at any time, but only has an effect if the browser is currently in the process of navigating to a new location. A canceled navigation will fire the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event and the event handler's *wParam* value will be FALSE. The navigation status will also include the `WEBVIEW_STATUS_NAVIGATION_CANCELED` bitflag which can be used to distinguish between a navigation that was canceled and one that failed for another reason.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetNavigationStatus, Navigate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::ClearCache Method

```
BOOL ClearCache();
```

The **ClearCache** method removes visited pages from the cache.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will clear the browser cache for the current session. This can be useful if you wish to force the browser to reload content. If you have created the browser instance using a temporary data folder, or you have specified the `WEBVIEW_OPTION_ENABLE_PRIVACY` option, the browser session will always start with a new user profile and will not contain any cached content.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCookies, ClearHeaders, ClearHistory, Redraw, Reload

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::ClearCookies Method

```
BOOL ClearCookies(
   BOOL bAllCookies
);
```

The **ClearCookies** method deletes cookies for the current profile or session.

Parameters

bAllCookies

An optional boolean value which specifies if all cookies for the current user profile should be deleted, or only those cookies created during the current session. If this parameter is a non-zero value then all cookies stored in the current user's profile will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method deletes cookies that have been created by the current WebView profile. If the *bAllCookies* parameter is set to TRUE, all stored cookies for every site are removed; if set to FALSE, only the cookies associated with the current session are cleared. The effect is comparable to clearing cookies in a regular browser: login sessions, preferences, and other state information stored in cookies will no longer be available. This only affects the WebView's private user data folder and does not modify cookies stored by the user's standalone Edge browser.

If you have used the WebView browser to authenticate with one or more websites, calling this method will typically log you out of those sites and require you to re-authenticate. If you have created the browser instance using a temporary data folder, or you have specified the `WEBVIEW_OPTION_ENABLE_PRIVACY` option, no cookies will be preserved after the session ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCache, ClearHeaders, ClearHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::ClearHeaders Method

```
BOOL ClearHeaders(
   DWORD dwHeaderType
);
```

The **ClearHeaders** method clears the current request and/or response headers for the current session.

Parameters

dwHeaderType

An unsigned integer value which may be one of the following values.

Value	Description
WEBVIEW_HEADERS_REQUEST	The custom request headers for the current session should be deleted. These are the HTTP headers which can be set or changed prior to the browser navigating to a new page. The standard request headers included with every request will continue to be sent.
WEBVIEW_HEADERS_RESPONSE	The response headers for the current session should be deleted. These are the headers which are returned by the server after navigation has completed. These headers are automatically deleted with each new navigation, making this option unnecessary in most cases.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method can be used to delete the custom request headers set with **SetHeader** or clear the response headers returned by the previous navigation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCache, ClearCookies, ClearHistory, GetHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::ClearHistory Method

```
BOOL ClearHistory();
```

The **ClearHistory** method clears the navigation history for the current session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method clears both the profile's stored browsing history and the navigation history for the current WebView instance. Internally, it uses the WebView2 profile APIs to remove stored history and related download entries, and then resets the back/forward navigation stack. The effect is similar to selecting clearing the browser history in a browser, where visited pages are removed from the user's history, and the in-memory list of sites you can go back or forward to is discarded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCache, ClearCookies, ClearHeaders, GoBack, GoForward

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Close Method

```
BOOL Close();
```

The **Close** method closes the window and releases all browser resources.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This method closes a browser window which was created using the **Create** method. It is recommended that you always use this method to close the window rather than posting a `WM_CLOSE` message because it ensures resources for the window, such as its icons, are released correctly. The process of destroying the window does not occur immediately and when this function returns it is not guaranteed the browser child window and its associated processes have completely stopped. Use the **WaitUntilClosed** method if you wish to wait for the cleanup process to complete.

This method will cause the `WEBVIEW_EVENT_WINDOW_DETACHED` event to fire as part of the normal process of destroying the window. If you implement a handler for this event, you should never delay this process by attempting to show a dialog or request input from the user (for example, by displaying a message box). At the point this event is fired, the browser objects have already been disposed and there is no way to prevent the process from completing.

If this method is not explicitly called, the class destructor will close the hosting window and release the resources allocated for the browser.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Create, Detach, WaitUntilClosed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Create Method

```
BOOL Create(
   LPCTSTR lpszUrl
);

BOOL Create(
   LPCTSTR lpszTitle,
   DWORD dwOptions,
   LPCTSTR lpszDataFolder
);

BOOL Create(
   LPCTSTR lpszTitle,
   DWORD dwStyle,
   LPRECT lpPosition,
   DWORD dwOptions,
   LPCTSTR lpszUrl,
   LPCTSTR lpszLanguage,
   LPCTSTR lpszDataFolder
);
```

The **Create** method creates a new window which hosts an instance of the browser control.

Parameters

lpszTitle

A null-terminated string which specifies the text displayed in the title bar. This parameter may be NULL or an empty string, in which case the title bar is set to the title of the document currently being displayed in the browser and will be automatically updated with subsequent navigations.

dwStyle

An unsigned integer which specifies the window style. This is the same windows style used with the **Create** function. If the value is zero, it will default to using the `WS_OVERLAPPEDWINDOW` style which creates an overlapped window with a caption, system menu and a minimize and maximize button. Some style bitflags will be ignored, such as the `WS_CHILD` style which cannot be applied to this window. This method can only be used to create an application window.

lpPosition

A pointer to a **RECT** structure which defines the position and size of the window. If all structure member values are `>= 0`, their values are passed directly to **Create**. If the top or left values are -1, a default position will be used. For example, if the structure is defined as `{ -1, -1, 800, 600 }` it would create an 800 x 600 window at a default position on the display. If the left or right values are -1, default values are used for the width or height. If all values are negative, the position and size of the window will use default values. If this parameter is NULL, the window will be positioned in the center of the current display with a width and height set to half the display area.

dwOptions

One or more option bitflags. For more information, refer to the Options documentation. Most applications should not use a value of zero for this parameter. If no options are specified, the browser instance will be created with a very restrictive set of permissions that can prevent many websites from functioning correctly.

lpszUrl

A null-terminated string which specifies the initial URL the browser should navigate to after the

window has been created. If this parameter is NULL or an empty string, the browser instance will be created with a blank page. The only schemes permitted are `https://`, `http://` and `file://`. If a domain name is specified without a scheme, it will default to using `https://`.

lpszLanguage

A null-terminated string which specifies an alternate language to use when the browser displays a context menu or dialogs. This value can be a standard language tag in BCP-47 format (such as "en-US") or it can be a string which specifies a hexadecimal Windows locale identifier (such as "0x0409"). It is recommended you use the language tag whenever possible, as Microsoft is deprecating the use of locale identifiers (LCIDs) and their associated APIs. If this parameter is a NULL pointer or an empty string, the browser will use the current default language for the local user.

lpszDataFolder

A folder name where the user profile data will be stored. This is the location where the browser will cache viewed pages, store cookies and form data, and where scripts may store session data. This value may be just a name (such as the name of the application) or an absolute path. If the folder name contains environment variables surrounded in percent (%) symbols, they will be replaced with their value in the current user's environment. If folder name does specify an absolute path, the profile will be stored in the current user's LocalAppData folder. If this parameter is NULL or an empty string, a temporary profile will be created for the session and then automatically deleted when the session ends.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method creates a top-level host window and a WebView runtime on a dedicated UI thread owned by the library. It returns after the window and control have been created and any navigation to the initial URL proceeds asynchronously. The returned window handle can be managed with normal Win32 APIs.

This method requires the single-threaded apartment (STA) COM model. During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), a browser instance cannot be created. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

The dwOptions parameter enables you to specify what features are enabled in the browser. It is recommended that most applications use `WEBVIEW_OPTION_ENABLE_DEFAULT` which enables functionality which is most commonly found in web browsers. If you want your host application to exchange messages with scripts executing in the browser, you should also specify the `WEBVIEW_OPTION_ENABLE_MESSAGES` options. This option must be enabled to use **PostWebMessage** and **GetWebMessage** methods.

When clicking on a link would normally open that page in a new tab or browser window, it will be redirected to the primary browser window by default. This ensures that the browser control retains focus and all events are within the context of that window. If you wish to permit the creation of popup windows and/or support the creation of new browser windows (for example, if a link target is "_blank") then specify the `WEBVIEW_OPTION_ENABLE_NEW_WINDOWS` option. Note that if you permit the creation of new popup windows, it can shift focus away from your main browser window. There is no mechanism in the API to enumerate the child popup windows and they are independent of the window you have created.

By default, the browser will use the language associated with the current user's locale. If you specify an alternate language, this also affects what languages the browser includes with the **Accept-Language** request header. This will provide hints to the website as to what language is preferred when it returns content to the browser. However, it does not guarantee that a website will return content in that language and will not automatically cause a page to be translated from one language to another.

If you pass NULL or an empty string for the *lpszDataFolder* parameter, the library creates a temporary, per-session user-data folder under the current user's `LocalAppData` folder. That folder is flagged as temporary and is automatically deleted when you detach the WebView from the window using the **Detach** method. If you pass a relative path (such as "MyApp"), the library creates or uses that folder beneath `%LocalAppData%\WebView`, creating a folder path like `%LocalAppData%\WebView\MyApp`.

If you pass a fully qualified path as the *lpszDataFolder* parameter, it will use that path. However, it will not accept paths to unsafe locations and will return an error if your application does not have permission to create a folder in that location. If there are environment variables surrounded by percent (%) symbols, those variables will be expanded when the folder path is resolved. If you enable privacy mode with `WEBVIEW_OPTION_ENABLE_PRIVACY`, the library will ensure the profile is unique to the session by appending a GUID to the base folder and treating it as temporary. The **GetUserDataFolder** method can be used to obtain the full path to the data folder being used.

When multiple WebView instances point at the same user-data folder, they share a single WebView2 profile created with that folder path. In this configuration the browser cache, cookies, history, and site storage are persisted under that folder and are reused across windows and processes, which speeds up subsequent navigations and preserves sign-in/session state. The library passes your folder directly to WebView2 when creating the environment, so any instance using that same path participates in the same profile, and you can clear portions of that profile at runtime via the provided helpers for cache, history, and cookies.

Event callbacks are invoked on the thread that owns the browser window, not the thread that calls **Create**. This is typically the internal UI thread that runs the message loop for the control. Avoid updating shared globals from both your main thread and the browser thread without synchronization. If you need to communicate back to your main UI, use **PostMessage** or **SendMessage** to send application-defined messages to the window, or use an event, critical section, or message queue rather than directly changing the shared state.

The method can be used from a console application. You do not need to create a message pump on your main thread; the library's hosting thread runs the necessary loop for the browser window. GUI applications can also call this method to create a modeless browser window that you control like any other top-level window.

Example

```
#include "cstools12.h"
#include <stdio.h>

// WebView event handler for navigation and security events
static VOID CALLBACK WebViewEvents(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    switch (uEvent)
    {
    case WEBVIEW_EVENT_NAVIGATION_STARTING:
        OutputDebugString(_T("[WebView] Navigation starting\n"));
        break;
```

```
    case WEBVIEW_EVENT_NAVIGATION_COMPLETED:
        if (wParam)
            OutputDebugString(_T("[WebView] Navigation completed\n"));
        else
            OutputDebugString(_T("[WebView] Navigation failed\n"));
        break;

    case WEBVIEW_EVENT_SECURITY_CHANGED:
        OutputDebugString(_T("[WebView] Security changed\n"));
        break;
    }
}

int _tmain(int argc, TCHAR *argv[])
{
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Unable to initialize the WebView API, error 0x%08lx\n"),
WebViewGetLastError());
        return 1;
    }

    // Create a top-level browser window
    LPCTSTR pszAppTitle = _T("MyWebView");
    LPCTSTR pszInitialUrl = _T("https://example.com");
    LPCTSTR pszDataFolder = _T("MyTestApp");
    DWORD dwOptions = WEBVIEW_OPTION_ENABLE_DEFAULT;
    RECT rcPos = { -1, -1, 1200, 800 };

    CWebView webViewer;
    BOOL bCreated = webViewer.Create(
                            pszAppTitle,          // Window title
                            WS_OVERLAPPEDWINDOW,  // Window style
                            &rcPos,               // Initial position
                            dwOptions,            // Creation options
                            pszInitialUrl,        // Initial URL
                            NULL,                 // Default language
                            pszDataFolder);       // User data folder
    if (!bCreated)
    {
        _tprintf(_T("Unable to create a WebView window\n"));
        return 2;
    }

    // Enable event notifications
    webView.EnableEvents(WebViewEvents, reinterpret_cast<DWORD_PTR>(&webView));

    TCHAR szCmdLine[1024];
    TCHAR *pszCmd;
    TCHAR *pszArg;

    // Read commands from standard input to control the browser
    while (pszCmd = _getts_s(szCmdLine, _countof(szCmdLine)))
    {
        // Trim any leading whitespace
        while (*pszCmd && _istspace(*pszCmd))
            pszCmd++;

        // Look for any arguments to the command
```

```
            pszArg = _tcschr(pszCmd, _T(' '));
            if (pszArg)
                *pszArg++ = TCHAR('\0');

            if (_tcsicmp(pszCmd, _T("url")) == 0)
            {
                // The URL command will navigate to a new URL
                if (pszArg && _tcslen(pszArg) > 0)
                {
                    if (!webViewer.Navigate(WEBVIEW_NAVIGATE_DEFAULT, pszArg))
                    {
                         DWORD dwError = webViewer.GetLastError();
                         _tprintf(_T("Navigate failed with error 0x%08lx\n"), dwError);
                    }
                }
                else
                {
                    _tprintf(_T("This command requires a URL\n"));
                }
            }
            else if (_tcsicmp(pszCmd, "quit") == 0)
            {
                // Exit the loop, closing the browser window
                break;
            }
        }

        // Destroy the browser window and wait for it to close
        webViewer.Close();
        webViewer.WaitUntilClosed(INFINITE);

        return 0;
    }
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Attach, Create, Detach, WebViewEventProc, GetUserDataFolder

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Detach Method

```
BOOL Detach();
```

The **Detach** method releases the resources allocated for the browser session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to detach the browser instance from a window that it was previously attached to using either the **Attach** method. It is automatically called by the class destructor when the browser is no longer needed and the host window is being destroyed. If the browser was created using the **Create** method, calling this method has the same effect as calling the **Close** method.

This method will cause the **WEBVIEW_EVENT_WINDOW_DETACHED** event to fire as part of the normal process of detaching from the window. If you implement a handler for this event, you should never delay this process by attempting to show a dialog or request input from the user (for example, by displaying a message box). At the point this event is fired, the browser objects have already been disposed and there is no way to prevent the process from completing.

If there is a navigation in progress at the time this method is called, it will be canceled. If there are one or more file downloads in progress, the transfers will be aborted.

The current thread will block while it waits for the WebView2 browser process to shutdown gracefully. If the browser becomes non-responsive, this method will wait several seconds before forcibly terminating the child process and continuing with the cleanup process. If a temporary user profile was specified or a private browsing session was created, this method will also clear the browser cache and remove the user data folder that was created for the session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Close

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::DisableEvents Method

```
BOOL DisableEvents();
```

The **DisableEvents** method disables event handling for the browser.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method removes the currently installed event handler for the browser and clears the user parameter. After it returns, the application will not receive any further browser events until **EnableEvents** is called again with a valid handler. Use this to permanently stop participating in event notifications (for example, during shutdown or when detaching the browser control from the host). Unlike temporarily freezing events, this method completely clears the callback handler for the browser instance.

If you only need to pause notifications temporarily, such as when presenting a modal dialog, performing a batch of UI updates, or executing several programmatic navigations back-to-back, freezing events is the recommended approach. Disabling events should be reserved for longer-term or terminal states, typically when the application is in the process of closing down.

You should not call **DisableEvents** from inside an event handler. Disabling callbacks while you are actively processing an event can lead to subtle reentrancy issues and missed cleanup. If you need to suspend notifications from within an event, call **FreezeEvents** instead, complete your work, and then unfreeze when you are ready to resume event handling.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

EnableEvents, FreezeEvents, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::EnableEvents Method

```
BOOL EnableEvents(
   WEBVIEWEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **EnableEvents** method enables event handling for the browser.

Parameters

lpEventProc

A callback function which will be invoked whenever browser events occur. This parameter cannot be NULL. The callback function will be invoked in the context of a separate thread which hosts the browser control, not the thread which calls this method. For more information, refer to the WebViewEventProc and Events documentation.

dwParam

An optional parameter which can be used to provide additional context to the event handler. This user-defined value is passed back to the callback function when an event occurs. If this information is not required, this parameter can be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method assigns or replaces the callback function that receives browser events for the specified host window. It can be used after the browser has been attached to start receiving notifications or to change event handlers at runtime. The change takes effect immediately and subsequent events will be invoked using the new handler.

Event delivery is serialized by the library and this call safely updates the event handler. You cannot specify a NULL pointer as the event handler; if you need to disable events, use the **DisableEvents** method. If events were previously frozen with **FreezeEvents**, calling this method with a new event handler will automatically unfreeze event notifications.

When using this method from within a C++ class, the event callback must be declared as a static function with the **__stdcall** (or **CALLBACK**) calling convention. This is required because the library calls the handler just as it would with any Win32 API callback function, without knowledge of C++ instance objects. Non-static member functions include a hidden **this** pointer parameter, which makes their signature incompatible with the expected **WEBVIEWEVENTPROC** type. A common pattern is to pass the object's ***this*** pointer as the ***dwParam*** value, then cast it back inside the static callback function to access the instance. This preserves object-oriented design while still meeting the calling convention requirements of the API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

DisableEvents, FreezeEvents, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::EnumDownloads Method

```
INT EnumDownloads(
   LPDWORD lpdwFileList,
   INT nMaxFiles
);

INT EnumDownloads(
   DWORD dwStatus,
   LPDWORD lpdwFileList,
   INT nMaxFiles
);
```

The **EnumDownloads** method enumerates the file identifiers for downloads during the current session.

Parameters

dwStatus

A value which specifies the status of the downloads to enumerate. If this value is zero, the method will enumerate all downloads. If this parameter is one of the values listed below, only those downloads which match the specified status will be enumerated. For example, if this parameter is **WEBVIEW_DOWNLOAD_COMPLETED** then only those file downloads which have completed will be enumerated.

Name	Description
WEBVIEW_DOWNLOAD_STARTED	The file download has started and is in-progress.
WEBVIEW_DOWNLOAD_COMPLETED	The file download has completed successfully.
WEBVIEW_DOWNLOAD_CANCELED	The file download has been canceled.
WEBVIEW_DOWNLOAD_FAILED	The file download failed due to a network error or other problem.

lpdwFileList

A pointer to an array of DWORD values which specify the unique file identifier associated with the download. The identifiers can be passed to the **GetDownloadInfo** method to get detailed information about a specific download. This parameter can be NULL, in which case it will be ignored and the method will return the number of downloaded files.

nMaxFiles

The maximum number of file identifiers which can be copied into the *lpdwFileList* array. If the *lpdwFileList* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of files downloaded during the current session. If there have been no downloads, or if the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnumDownloads** method can be used to obtain the identifiers for one or more file downloads that have occurred during the current browser session. Each identifier returned by this method uniquely corresponds to a download and can be used with functions such as **GetDownloadInfo** or

GetDownloadPath to obtain detailed information about the file.

If the *dwStatus* parameter is set to zero, the method enumerates all downloads regardless of their state. Otherwise, only downloads that match the specified status value are included in the enumeration. This allows applications to filter the results to include only downloads that are currently in progress, those that completed successfully, or those that were canceled or failed.

When the *lpdwFileList* parameter is not NULL, the method copies up to *nMaxFiles* identifiers into the specified array. If the number of matching downloads exceeds the array size, only the first *nMaxFiles* identifiers are returned, but the return value will still reflect the total number of matching downloads. This enables the application to first call the method with a NULL pointer to determine how many downloads are available, and then allocate an appropriately sized buffer before retrieving the identifiers.

The enumeration includes both active and completed downloads. If the host window is detached or destroyed, the list of downloads is cleared and further enumeration will return zero results.

Example

```
// Get the total number of downloads
INT nTotal = pWebView->EnumDownloads();

if (nTotal <= 0)
{
    _tprintf(_T("No downloads found.\n"));
    return;
}

// Allocate a buffer for the identifiers
DWORD *pIds = (DWORD*)malloc(nTotal * sizeof(DWORD));
if (!pIds)
    return;

INT nCount = pWebView->EnumDownloads(pIds, nTotal);

// Enumerate each download and print basic info
for (INT nIndex = 0; nIndex < nCount; nIndex++)
{
    WEBVIEW_DOWNLOAD_INFO fileInfo = { sizeof(fileInfo) };

    if (pWebView->GetDownloadInfo(pIds[nIndex], &fileInfo))
    {
        _tprintf(_T("Download %u: %s (%llu/%llu bytes)\n"),
                 fileInfo.dwFileId,
                 fileInfo.szFileName,
                 fileInfo.uiBytesCopied.QuadPart,
                 fileInfo.uiBytesTotal.QuadPart);
    }
}

free(pIds);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

CancelDownload, GetDownloadInfo, GetDownloadPath, SetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Execute Method

```
INT Execute(
   LPCTSTR lpszJavaScript,
   LPTSTR lpszResult,
   INT nMaxLength
);
```

The **Execute** method executes JavaScript and returns the result synchronously.

Parameters

lpszJavaScript

A null-terminated string which specifies the JavaScript that should be executed in the context of the current page being viewed. This parameter is required and cannot be NULL.

lpszResult

A string buffer which will contain the result returned by the script, either as a JSON object or a JSON string value. If this parameter is NULL, the method will not copy the result but will return the total number of characters in the response, not including the terminating null.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null. If the ***lpszResult*** parameter is NULL, this parameter should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the result buffer, not including the terminating null. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Execute** method evaluates a JavaScript expression within the context of the currently loaded page and returns the result synchronously. Unlike **AsyncExecute**, this method blocks the current thread while waiting for the script to complete. If the web page is non-responsive, this method will timeout after approximately two seconds.

If you wish to dynamically allocate a string buffer for the response, you can call this method with the ***lpszResult*** parameter as NULL and the ***nMaxLength*** parameter as zero. The function will return the number of characters in the response, not including the terminating null. You can allocate the buffer and then call **GetWebResult** which will copy the response into the buffer you provide. Because script execution is dispatched to the underlying WebView2 control, this method will fail if navigation is still in progress. Applications should wait for the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event before invoking script execution.

Example

```
// Execute a simple script that returns a string
INT nLength = pWebView->Execute(_T("document.title"), NULL, 0);

if (nLength > 0)
{
    // Allocate a buffer large enough to hold the result
    LPTSTR lpszResult = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (lpszResult)
    {
```

```
        // Retrieve the result of the script execution
        if (pWebView->GetWebResult(lpszResult, nLength + 1))
        {
            _tprintf(_T("Page title is: %s\n"), lpszResult);
        }
        free(lpszResult);
    }
}
else
{
    _tprintf(_T("Script execution failed\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AsyncExecute, GetWebResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::FindBrowser Method

```
HWND FindBrowser(
   HWND hWndParent,
   LPCTSTR lpszSearch,
   DWORD dwFlags,
   LPDWORD lpdwThreadId
);
```

The **FindBrowser** method locates the associated host or child window handle for a browser instance.

Parameters

hWndParent

Handle to the parent window. If this parameter is NULL, it will search all top-level windows. If this parameter is not NULL, it must be a valid window handle and the method will only match against a window which is a child window of the specified handle.

lpszSearch

An optional null-terminated string which specifies a URL. The function will only match those browser windows which have navigated to the specified URL. A simple case-insensitive substring search is performed, but will only match windows created by the current process. If this parameter is NULL or an empty string, the method will not match against any specific URL.

dwFlags

This parameter is reserved for future use and should always be zero. Future versions may expand this to support additional types of matching, such as document titles or more advanced search expressions.

lpdwThreadId

An optional parameter which is a pointer to an unsigned integer value that will contain the thread ID for the browser instance if a matching window is found. If no browser is found, this value will be set to zero. If this information is not required, this parameter can be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **FindBrowser** method can be used to locate the window handle for an existing WebView2 instance that was created by the calling process. This is useful if the application needs to re-establish a reference to a previously created browser window, for example when handling messages or coordinating multiple windows.

If a parent window handle is specified, the method will restrict its search to that window and its children. This is the most efficient approach when the caller already knows the expected parent container. If the parameter is NULL, all top-level windows owned by the process are searched. An optional URL string can also be provided to limit results to windows whose current location contains the specified substring. The search is case-insensitive and only applies to windows created by the current process.

The method only returns a single matching window handle. If multiple instances could match the criteria, the particular instance returned is not guaranteed and the caller should design accordingly. When both the parent handle and search string are NULL, the method simply locates the first available

WebView2 instance owned by the current thread.

If the optional ***lpdwThreadId*** parameter is supplied, it will receive the thread identifier that owns the WebView2 instance. This can be used when thread affinity is important, such as posting messages directly to the UI thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Create, GetChildWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::FreezeEvents Method

```
BOOL FreezeEvents(
   BOOL bFreezeEvents
);
```

The **EnableEvents** method suspends or re-enables event notifications for the specified browser.

Parameters

bFreezeEvents

An integer value which specifies if event handling should be frozen. If the value is non-zero, browser events will be suspended until they are re-enabled. If the value is zero, browser events will be re-enabled if an event handler has been defined.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method toggles a per-instance switch for event delivery. When events are frozen, the browser continues to operate normally and still generates events internally, but your application will not be notified of them. The events are not queued while frozen; they are discarded. For developers familiar with COM controls, this mirrors the behavior of `IOleControl::FreezeEvents`. When you unfreeze, only new events that occur after unfreezing will be delivered.

Event freezing can be useful when:

- You are running scripted or programmatic operations that would otherwise flood your event handler, such as multiple page redirects or rapid **AsyncExecute** calls.
- You are making a series of user interface changes and want to avoid reentrant updates. For example, temporarily stopping focus and/or size change events while moving or resizing controls.
- You are showing a modal dialog or performing work where event reentrancy could complicate state management.
- You need to suppress event-driven logic during controlled navigation, such as "silently" reloading a document or navigating to a new location.

Freezing does not alter browser state, navigation, or timers; it only affects your application's callback. To resume normal notifications, call this method with the *bFreezeEvents* parameter as zero. Freezing events will not affect calls to **WaitForEvent**. Because freezing events does not prevent browser events from occurring, they are still processed internally and you can wait for a specific event to complete.

If you discover that subsequent notifications within an event handler would interfere with your application's workflow, it is safe to freeze events from within the handler. After completing the operation, unfreeze events (for example, by posting an application-defined message or using a timer) so events resume normally. You should avoid calling **DisableEvents** from within an event handler, use freezing instead.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

DisableEvents, EnableEvents, WebViewEventProc, WaitForEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetAllOptions Method

```
BOOL GetAllOptions(
   LPDWORD lpdwOptions
);
```

The **GetAllOptions** method returns the current option flags for the instance.

Parameters

lpdwOptions

A pointer to an unsigned integer which will contain the browser options for the current session. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns the options for the current session. For more information about the options available, review the Options documentation. Always check the return value from this method; the option bitflags are only valid if this method returns a non-zero (TRUE) value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, GetOption, SetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetBackgroundColor Method

```
BOOL GetBackgroundColor(
   COLORREF *lprbgBackground
);
```

The **GetBackgroundColor** method returns the background color used behind the page content.

Parameters

lprbgBackground

A pointer to an integer which will contain the current background color as an RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The background color specifies the fill used by the browser window when there is no content to display or the document itself does not define a background. This value only applies to the WebView host surface and does not affect page transparency. The alpha channel is ignored, and the color is treated as fully opaque. By default, the background color for new browser instances will be white.

A **COLORREF** is a 32-bit value used by the Windows API to represent an RGB color. The low-order byte specifies the red component, the next byte the green component, and the third byte the blue component, in the format **0x00BBGGRR**. The high-order byte is unused and should be zero. Note that **COLORREF** is specific to the Windows API and is not the same as the OLE color system (**OLE_COLOR**). OLE uses a slightly different encoding that allows system color values to be specified, which are resolved at runtime. Developers should be careful not to confuse the two.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

SetBackgroundColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetBrowserVersion Method

```
INT GetBrowserVersion(
   LPDWORD lpdwVersionMS,
   LPDWORD lpdwVersionLS,
   LPTSTR lpszVersion,
   INT nMaxLength
);
```

The **GetBrowserVersion** method returns the browser runtime version information.

Parameters

lpdwVersionMS

A pointer to an unsigned integer which contains the major and minor version numbers. This parameter may be NULL.

lpdwVersionLS

A pointer to an unsigned integer which contains the build and release version numbers. This parameter may be NULL.

lpszVersion

A null-terminated string buffer which will contain the version as a printable string. This parameter may be NULL, in which case the version string is not copied but the method will still return the length of the version string if successful. If this parameter is NULL, the *nMaxLength* parameter should have a value of zero.

nMaxLength

An integer which specifies the maximum number of characters which may be copied into the string buffer, including the terminating null. If the *lpszVersion* parameter is NULL, this parameter should be zero.

Return Value

If the method succeeds, the return value is the length of the version string, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetBrowserVersion** method returns the version information for the installed Microsoft Edge WebView2 Runtime. This is the runtime that hosts the embedded browser control, and it is separate from the version of the Edge browser that may be installed on the system. The method can provide the version number as two 32-bit values (major/minor and build/revision), and optionally as a formatted string for display purposes. This method is useful for diagnostics, logging, or when an application requires a specific minimum runtime version to support certain features. If the WebView2 Runtime is not installed or cannot be located, the method will fail and return zero.

The version values returned by *lpdwVersionMS* and *lpdwVersionLS* can be used with the `HIWORD` and `LOWORD` macros to extract the individual components of the version number (for example, major, minor, build, and revision). The printable version string will match the same four-part format commonly shown in Edge and WebView2 documentation (e.g., "`126.0.2592.113`").

The result is independent of whether a WebView window has already been created or attached. It only reflects the version of the installed runtime and does not indicate whether the runtime is currently in use. For cases where you only need to check if a runtime is available, use the **RuntimeAvailable**

method instead.

Example

```
DWORD dwVerMS = 0, dwVerLS = 0;
TCHAR szVersion[64] = {0};

INT nLength = pWebView->GetBrowserVersion(&dwVerMS, &dwVerLS, szVersion,
_countof(szVersion));
if (nLength > 0)
{
    WORD wMajor = HIWORD(dwVerMS);
    WORD wMinor = LOWORD(dwVerMS);
    WORD wBuild = HIWORD(dwVerLS);
    WORD wRevision = LOWORD(dwVerLS);

    _tprintf(_T("WebView2 Runtime Version: %s\n"), szVersion);
    _tprintf(_T("Major=%hu Minor=%hu Build=%hu Revision=%hu\n"), wMajor, wMinor,
wBuild, wRevision);
}
else
{
    _tprintf(_T("Unable to determine WebView2 Runtime version (Error 0x%lx)\n"),
pWebView->GetLastError());
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Attach, Create, RuntimeAvailable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetChildWindow Method

```
HWND GetChildWindow();
```

The **GetChildWindow** method returns the handle of the internal browser child window.

Parameters

None.

Return Value

If the method succeeds, the return value is a window handle. If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The value returned by this method is the handle to the internal child window that hosts the WebView control. Applications should treat this handle as an implementation detail and avoid sending messages or altering window styles directly. The control manages this child window and relies on its state for proper operation. Interfering with the child window (for example, by subclassing it, changing its parent, or destroying it) can cause undefined behavior and application instability.

This method is primarily provided to support advanced scenarios where access to the native window handle is unavoidable, such as integrating with APIs that require a child window handle. In most cases, developers should use the higher-level functions in the **CWebView** class instead of interacting with the child window directly.

If you want to get the handle to the window hosting the browser control, use the **GetSafeHwnd** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, FindBrowser, GetSafeHwnd

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetContentType Method

```
INT GetContentType(
   LPTSTR lpszContentType,
   INT nMaxLength
);
```

The **GetContentType** method returns the content type for the current page.

Parameters

lpszContentType

A string buffer which will contain the MIME content type of the resource returned by the web server. The buffer must be large enough to contain the entire content type string including the terminating null character. If this parameter is NULL, the content type will not be copied but the method will return the length of the `Content-Type` header value.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer, including the terminating null. If the *lpszContentType* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the length of the `Content-Type` header value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetContentType** method returns the MIME type reported by the server in the `Content-Type` response header for the most recent navigation. This value reflects the server's indication of what kind of content is being delivered, such as "`text/html`", "`application/pdf`", or "`image/png`". The returned string is preserved until the next navigation begins, even if the document is cleared or headers are reset.

It is important to note that not all servers will provide a `Content-Type` header. Some intentionally omit this information in combination with a `X-Content-Type-Options: nosniff` header as a security measure to prevent the client from "sniffing" or inferring the type. In such cases, **GetContentType** will return an empty string. Applications should be prepared for this scenario and not assume a value will always be available.

This method is a convenience wrapper around the underlying response header and is useful for distinguishing whether the WebView is rendering standard HTML or other document types. If more detailed information is needed about the response, developers can enumerate the response headers directly using **GetFirstHeader** and **GetNextHeader**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHeaders, GetFirstHeader, GetHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetCurrentUrl Method

```
BOOL GetCurrentUrl(
   LPTSTR lpszUrl,
   INT nMaxLength
);
```

The **GetCurrentUrl** method returns the URL for the current document.

Parameters

lpszUrl

A string buffer which will contain the current URL when the function returns. If this parameter is NULL, the method will not copy anything into the buffer but will return the length of the URL, not including the terminating null character. If a buffer is provided, it must be large enough to contain the entire URL.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null. If the *lpszUrl* parameter is NULL this value should be zero.

Return Value

If the method succeeds, the return value is the length of the current URL, not including the terminating null. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetCurrentUrl** method retrieves the URL associated with the current document loaded in the browser instance. The URL value is available as soon as navigation begins to a new location, so it can be queried during or after the navigation process. This allows applications to obtain the destination address immediately, even before the page has fully loaded or rendered its content.

Note that the current URL may differ from the value originally passed to **Navigate**. The browser engine can normalize the address, such as adjusting letter case, expanding relative paths, or adding default components like a trailing slash. In addition, the current URL can change multiple times during the course of a navigation if the server issues one or more HTTP redirects before the final document is retrieved. For this reason, the URL returned by this method should be treated as the authoritative location of the content currently being displayed.

This method differs from examining the document itself (for example, by executing a script to query `document.URL`) because it reflects the navigation state maintained internally by the control. Applications should use this method to reliably track the current location of the browser instance.

If the **SetDocumentText** method was called to change the current document, the URL returned by this method will typically be `"data:text/html"` which indicates it is a local data source.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Navigate, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetDocumentLength Method

```
INT GetDocumentLength();
```

The **GetDocumentText** method returns the current document length.

Parameters

None.

Return Value

If the method succeeds, the return value is the length of the document text in characters. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDocumentLength** method returns the number of characters in the current document. This will reflect the state of the document at the time the method is called. If the page has been dynamically modified by scripts after it was initially loaded, those changes will be included. Note that if your project is configured to use Unicode, this value will not be the same as the size of the document in bytes.

This method should only be called after navigation has completed. If called while a page is still loading, the contents of the page may be incomplete or empty. The length of the document includes the HTML markup and inline scripting.

If a blank page is currently being displayed, such as when navigating to `about:blank` or by setting the current document text to an empty string, this method can still return a value. Internally, the browser creates a blank page using the markup `"<html><head></head><body></body></html>"` and this method will return the length of that content.

The **GetFilteredText** method can be used to obtain a filtered version of the text with the HTML markup removed.

Example

```
INT nLength = pWebView->GetDocumentLength();
if (nLength > 0)
{
    LPTSTR pszBuffer = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (!pszBuffer)
    {
        // Virtual memory exhausted
        return;
    }

    nLength = pWebView->GetDocumentText(pszBuffer, nLength + 1);
    if (nLength > 0)
    {
         // Write the document HTML to standard output
        _putts(pszBuffer);
        _puttc(_T('\n'), stdout);
    }

    free(pszBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDocumentText, GetDocumentTitle, GetFilteredText, Navigate, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetDocumentText Method

```
INT GetDocumentText(
   LPTSTR lpszText,
   INT nMaxLength
);
```

The **GetDocumentText** method returns the current document text.

Parameters

lpszText

A string buffer which will contain the current document text when the method returns. If this parameter is NULL, the method will not copy anything into the buffer but will return the length of the text, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null. If the *lpszText* parameter is NULL this value should be zero.

Return Value

If the method succeeds, the return value is the length of the document text in characters, not including the terminating null. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDocumentText** method retrieves the HTML markup for the current document loaded in the browser. This includes the complete contents of the `<html>` element, such as the `<head>` and `<body>` sections, rather than just the visible text.

The text returned will reflect the current state of the document at the time the method is called. If the page has been dynamically modified by scripts after it was initially loaded, those changes will be included. The value may be large, particularly for complex pages, and applications should allocate a buffer of sufficient size by first calling the method with a NULL pointer to determine the required length.

If your project is configured to use a multi-byte character set (ANSI) instead of Unicode, this method will always return the content as UTF-8 encoded text, regardless of the system's current locale or code page. Legacy ANSI code pages cannot reliably represent the full range of characters, and using them in place of Unicode or UTF-8 can lead to data loss or corrupted text.

This method should only be called after navigation has completed. If called while a page is still loading, the results may be incomplete or empty. When the content is provided by a local file or string, the returned HTML will be the original content that was passed to the control. The returned text is raw HTML and may contain scripts, style definitions, and markup that are not directly suitable for display in a plain-text context.

If a blank page is currently being displayed, such as when navigating to `about:blank` or by setting the current document text to an empty string, this method can still return a value. Internally, the browser creates a blank page using the markup `"<html><head></head><body></body></html>"` and that is what this method will return.

The **GetFilteredText** method can be used to obtain a filtered version of the text with the HTML markup removed.

Example

```
INT nLength = pWebView->GetDocumentLength();
if (nLength > 0)
{
    LPTSTR pszBuffer = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (!pszBuffer)
    {
        // Virtual memory exhausted
        return;
    }

    nLength = pWebView->GetDocumentText(pszBuffer, nLength + 1);
    if (nLength > 0)
    {
         // Write the document HTML to standard output
        _putts(pszBuffer);
        _puttc(_T('\n'), stdout);
    }

    free(pszBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDocumentLength, GetDocumentTitle, GetFilteredText, Navigate, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetDocumentTitle Method

```
INT GetDocumentTitle(
   LPTSTR lpszTitle,
   INT nMaxLength,
);
```

The **GetDocumentTitle** method returns the document title.

Parameters

lpszTitle

A string buffer which will contain the current document title when the method returns. If this parameter is NULL, the method will not copy anything into the buffer but will return the length of the title, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null. If the *lpszTitle* parameter is NULL this value should be zero.

Return Value

If the method succeeds, the return value is the length of the document title in characters, not including the terminating null. If the document does not have a title or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDocumentTitle** method returns the current title of the document loaded in the browser. This corresponds to the contents of the `<title>` element in the document's `<head>` section, if one is defined.

The title is not fixed at the time the page is first loaded. A script running in the document can modify the `<title>` element dynamically, and some web applications use this technique to reflect state changes (for example, to indicate new messages or status updates). Because of this, the value returned by this method may change over the lifetime of the page. If the document does not explicitly define a title, the method may return an empty string. Applications that display document titles should be prepared to handle this case.

If the browser window was created using **Create** without a specific title, the application title will be automatically changed to reflect the current document title. This does not happen if the browser window is attached to an existing window or if an application title was specified when the window was created. If you wish to manually update a top-level window title with the current document title, call this method in response to a `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event and then call the **SetWindowText** function to change the text in the window's title bar.

If the current document does not have a title, this method will return zero and the last error code will be set to `NO_ERROR`. If the *lpszTitle* parameter is not NULL, it will be set to an empty string when the method returns. This is not uncommon if the **SetDocumentText** method is called to change the current document and the text does not contain HTML markup or does not include a `<title>` tag.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDocumentText, Navigate, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetDownloadInfo Method

```
BOOL GetDownloadInfo(
   DWORD dwFileId,
   WEBVIEW_DOWNLOAD_INFO *lpInfo
);
```

The **GetDownloadInfo** method returns information about the status of the specified download.

Parameters

dwFileId

An unsigned integer value which uniquely identifies the download. This value cannot be zero.

lpInfo

A pointer to a `WEBVIEW_DOWNLOAD_INFO` structure which will contain information about the specified download when the method returns. The *dwSize* member of this structure must be initialized to the size of the structure prior to being passed to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns information about a specific download and is typically called in the context of a download event handler. The *dwFileId* parameter identifies the download to be queried. This identifier can be obtained either through download-related event notifications, where it is provided as the *lParam* value, or by calling the **EnumDownloads** method to enumerate active and completed downloads.

The file identifier should be treated as an opaque value. While it is guaranteed to be unique for the lifetime of the client session, the underlying implementation simply assigns sequential values. Applications must not depend on this behavior, as the specific numbering scheme may change in future versions. Because the identifiers are only unique within a single session, they cannot be used after the browser instance has been detached or destroyed. If you need to track downloads across sessions, you should use the file name or URL rather than the internal identifier.

Information is returned in the `WEBVIEW_DOWNLOAD_INFO` structure, which includes the download status, number of bytes copied, total size, elapsed time, file name, and the source URL. If the specified identifier does not correspond to a valid download, the function will fail.

If you are only interested in the name of the local file used for the specified download, you can use the **GetDownloadPath** function.

Example

```
// Get the total number of downloads
INT nTotal = pWebView->EnumDownloads();

if (nTotal <= 0)
{
    _tprintf(_T("No downloads found.\n"));
    return;
}
```

```
// Allocate a buffer for the identifiers
DWORD *pIds = (DWORD*)malloc(nTotal * sizeof(DWORD));
if (!pIds)
    return;

INT nCount = pWebView->EnumDownloads(pIds, nTotal);

// Enumerate each download and print basic info
for (INT nIndex = 0; nIndex < nCount; nIndex++)
{
    WEBVIEW_DOWNLOAD_INFO fileInfo = { sizeof(fileInfo) };

    if (pWebView->GetDownloadInfo(pIds[nIndex], &fileInfo))
    {
        _tprintf(_T("Download %u: %s (%llu/%llu bytes)\n"),
                 fileInfo.dwFileId,
                 fileInfo.szFileName,
                 fileInfo.uiBytesCopied.QuadPart,
                 fileInfo.uiBytesTotal.QuadPart);
    }
}

free(pIds);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CancelDownload, EnumDownloads, GetDownloadPath, SetDownloadPath, WEBVIEW_DOWNLOAD_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetDownloadPath Method

```
INT GetDownloadPath(
   DWORD dwFileId,
   LPTSTR lpszFilePath,
   INT nMaxLength
);
```

The **GetDownloadPath** method returns the complete path of the local file for the specified download.

Parameters

dwFileId

An unsigned integer value which uniquely identifies the download. This value cannot be zero.

lpszFilePath

A null-terminated string buffer which will contain the full path to the local file. If this parameter is NULL, the file path is not copied but the method will return the length of the file path, not including the terminating null character. It is recommended that this buffer be large enough to store `MAX_PATH` (260) characters.

nMaxLength

An integer value which specifies the maximum number of characters which may be copied into the string buffer, including the terminating null. If the ***lpszFilePath*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the length of the local file name, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns the complete path to the file that will be created or overwritten during the download. The file path returned by this method is always an absolute path which ensures that the application does not need to resolve relative paths or expand environment variables. The path returned may refer to a file that has not yet been created or fully written if the download is still in progress. Applications should check the download status with **GetDownloadInfo** before assuming the file is complete and ready to use.

The ***dwFileId*** parameter uniquely identifies the download within the current client session. This identifier can be obtained from download-related event notifications or by calling **WebViewEnumDownloads**. As with other functions that use file identifiers, applications should treat this value as opaque and only valid for the lifetime of the session.

If the provided string buffer is too small, the method will return the required length of the file path, not including the terminating null character. The caller can use this information to allocate a sufficiently large buffer and call the method again. There are both ANSI and Unicode versions of this method and the correct version is called based on how the project is configured. If the **GetDownloadPathA** method is called, the file path will always be returned as a UTF-8 encoded string regardless of the current locale.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CancelDownload, EnumDownloads, GetDownloadInfo, SetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetFilteredText Method

```
INT GetFilteredText(
   LPTSTR lpszFilteredText,
   INT nMaxLength
);
```

The **GetFilteredText** method filters HTML formatting from the current document and returns plain text in a string buffer.

Parameters

lpszFilteredText

A null-terminated string buffer which will contain the filtered text. If this parameter is NULL and the *nMaxLength* parameter is zero, the method will return the number of characters required to store the filtered text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the *lpszFilteredText* string buffer. If this value is zero, no characters will be copied to the buffer and the method will return the length of the filtered text, not including the terminating null character. This can be used to determine how much memory should be allocated for the string.

Return Value

If the method succeeds, it will return the number of characters copied into the string buffer, not including the terminating null character. If the *lpszFilteredText* parameter is NULL and *nMaxLength* is zero, the method will return the number of characters in the filtered text without copying any characters. If the method fails, or there is no text in the input buffer, the method will return zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFilteredText** method filters the HTML markup from the document text and returns output suitable for display in text-only environments. HTML tags are removed and entities are converted to their canonical Unicode equivalent. When filtering markup from the text, the method will:

- Remove all leading and trailing whitespace from the text, including newlines.
- Remove script, style, and other non-visible or unsafe elements.
- Convert common tag structures, such as lists and headers, into readable plain text.
- Extract and filter hyperlinks and image alt text, ignoring internal links to other sections.
- Remove any embedded credentials in hyperlinks.
- Decode common HTML entities such as `&trade` and `&copy`, as well as numeric entities like `&#x20.`

The method will remove extraneous whitespace within the text, but will attempt to maintain the structure of ordered and unordered lists. It does not parse the style information or Javascript that may be included in the HTML content. If a paragraph uses a style that would normally prevent it from being rendered in a browser (e.g.: the visibility attribute set to "hidden"), it is ignored and the text will be included in the output. If there are sections which contain preformatted text, such as text inside `<pre>...</pre>` or `<code>...</code>` tags, individual line and paragraph breaks will be preserved.

This method will not remove Unicode characters from the input text. If the HTML markup includes any entities which are represented as Unicode, such as which represents "½", they will be

```
&frac12;
```

returned in the output string buffer as Unicode characters. If the ANSI version of this method is called, Unicode will always be returned as UTF-8 encoded characters.

Example

```
INT nLength = pWebView->GetFilteredText(NULL, 0);
if (nLength > 0)
{
    LPTSTR pszFiltered = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (!pszFiltered )
    {
        // Virtual memory exhausted
        return;
    }

    nLength = pWebView->GetFilteredText(pszFiltered, nLength + 1);
    if (nLength > 0)
    {
         // Write the plain text to standard output
        _putts(pszFiltered);
    }

    free(pszFiltered);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDocumentText, PrintDocument, SaveDocument, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetFirstHeader Method

```
BOOL GetFirstHeader(
   LPTSTR lpszHeaderName,
   INT nMaxNameLen,
   LPTSTR lpszHeaderValue,
   INT nMaxValueLen
);
```

The **GetFirstHeader** method begins enumerating response headers and returns the first name/value pair.

Parameters

lpszHeaderName

A string buffer which will contain the name of the first header returned by the server. If this parameter is NULL it will be ignored.

nMaxNameLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderName* string buffer, including the terminating null. If the *lpszHeaderName* parameter is NULL this value should be zero.

lpszHeaderValue

A string buffer which will contain the value of the first header returned by the server. If this parameter is NULL, it will be ignored.

nMaxValueLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderValue* string buffer, including the terminating null. If the *lpszHeaderValue* parameter is NULL this value should be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The response headers exposed by this method correspond to the most recent completed navigation for the specified browser instance. Callers typically use this method to start an enumeration and then call **GetNextHeader** repeatedly until it returns FALSE to indicate there are no more headers to enumerate.

Header names are case-insensitive by definition. This method will usually return header names as lower-case values, but applications should never depend on the casing and should compare names case-insensitively. The header names and values are returned exactly as enumerated internally by the WebView2 control.

Some response headers can appear multiple times in a message. For list-valued headers (those defined by the HTTP specifications to be comma-joinable), the implementation may return a single entry with a comma-separated value (e.g., `Accept-Language`, `Cache-Control`, etc.). Other headers must not be combined; for example, each `Set-Cookie` header is returned as a separate name/value pair and will be seen multiple times during enumeration, each with a single cookie value. Code that depends on specific headers should be prepared for either a single combined value or multiple discrete occurrences of the same header name, depending on the header's semantics.

Enumeration reflects a snapshot of the headers for that response. Beginning a new navigation or calling **ClearHeaders** invalidates the current enumeration. Only one header enumeration can be active per browser instance at a time. The enumeration state is not thread-safe, and any access to the response headers should be synchronized across different threads.

If the provided buffers are too small, the header names and values will be truncated to the available space specified by the maximum length parameters. In most cases, a header name is not going to exceed 256 characters and a header value will normally not exceed 5120 characters. Most header values are relatively short, but some headers such as **Set-Cookie** can return much longer values.

This method cannot be called while the browser is in the process of navigating to a new location. It will also fail if the current document is a local file or if the document text was set dynamically using **SetDocumentText**. There are no response headers available in a local context.

Example

```
TCHAR szName[WEBVIEW_MAX_HEADER_NAME];
TCHAR szValue[WEBVIEW_MAX_HEADER_VALUE];

if (pWebView->GetFirstHeader(szName, _countof(szName),
                             szValue, _countof(szValue)))
{
    do
    {
        _tprintf(_T("%s: %s\n"), szName, szValue);
    }
    while (pWebView->GetNextHeader(szName, _countof(szName),
                                   szValue, _countof(szValue)));
}
else
{
    DWORD dwError = pWebView->GetLastError();
    _tprintf(_T("No headers available (error 0x%lx)\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHeaders, GetHeader, GetNextHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetFocus Method

```
BOOL GetFocus(
   BOOL *lpbHasFocus
);
```

The **GetFocus** method returns whether the browser has focus.

Parameters

lpbHasFocus

A pointer to a BOOL which will contain TRUE (non-zero) if the browser window has focus and FALSE (zero) if it does not have focus. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFocus** method queries whether the browser control currently has input focus. This is useful for determining if keyboard input will be directed to the content area of the browser or handled by the host window. The focus state can change dynamically as the user interacts with the application, and applications that track focus should also monitor the `WEBVIEW_EVENT_FOCUS_CHANGED` event.

This method only reports the focus state of the embedded browser, not of the host window itself. For example, a host window may be the active foreground window, but if another control within the host has focus, the browser will not be considered focused. To explicitly give focus to the browser, call **SetFocus**.

Example

```
BOOL bHasFocus = FALSE;

if (pWebView->GetFocus(&bHasFocus))
{
    if (!bHasFocus)
    {
        // Give the browser control focus if it does not have it
        pWebView->SetFocus();
    }
}
else
{
    // Handle error
    DWORD dwError = pWebView->GetLastError();
    _tprintf(_T("GetFocus failed, error=0x%lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Redraw, IsVisibile, SetFocus, SetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetHeader Method

```
INT GetHeader(
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue,
   INT nMaxLength
);
```

The **GetHeader** method returns the value of a specific response header by name.

Parameters

lpszHeaderName

A null-terminated string which specifies the response header name. This value is not case-sensitive but must match the complete header name. This parameter cannot be NULL.

lpszHeaderValue

A string buffer which will contain the value of the specified header. If this parameter is NULL then nothing will be copied but the method will return the length of the header value, not including the terminating null.

nMaxLength

The maximum number of characters which may be copied into the string buffer, including the terminating null. If the *lpszHeaderValue* parameter is NULL this value should be zero.

Return Value

If the method succeeds, the return value is the length of the header value, not including the terminating null. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetHeader** method retrieves the value of a specific HTTP response header returned by the server. Header names are matched in a case-insensitive manner, with leading and trailing whitespace automatically trimmed before comparison. If multiple values are present for the specified header name, the behavior depends on the type of header:

- For headers that must not be combined (such as `Set-Cookie` and `Proxy-Authenticate`), only the first value encountered is returned.
- For all other headers, multiple values are joined together and returned as a single comma-separated string, as defined by the HTTP specification.

For headers such as `Set-Cookie`, which may legitimately occur multiple times with distinct values, **GetHeader** will only return the first occurrence. If your application needs to process each value individually, you should use **GetFirstHeader** and **GetNextHeader** to enumerate all response headers rather than relying on this method.

If the *lpszHeaderValue* parameter is NULL, the method returns the number of characters required to store the complete header value, excluding the terminating null. This enables applications to dynamically allocate a buffer of the correct size before retrieving the actual value.

Response headers are only available after navigation has completed. Calling this method while a navigation is still in progress, or when displaying local file or data content, will fail.

The maximum recommended length for a header name is `WEBVIEW_MAX_HEADER_NAME` (256

characters) and for a header value is **WEBVIEW_MAX_HEADER_VALUE** (5120 characters). These limits are intended as practical guidelines, not absolute restrictions.

Example

```
 INT nLength = WebViewGetHeader(hWndWebView, _T("Content-Type"), NULL, 0);
if (nLength <= 0)
{
    _tprintf(TEXT("Unable to get Content-Type header (error 0x%lx)\n"),
WebViewGetLastError());
    return;
}

LPTSTR lpszValue = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
if (!lpszValue)
{
    _tprintf(_T("Out of memory\n"));
    return;
}

if (WebViewGetHeader(hWndWebView, _T("Content-Type"), lpszValue, nLength + 1) > 0)
{
    _tprintf(_T("Content-Type: %s\n"), lpszValue);
}
else
{
    _tprintf(_T("Failed to retrieve Content-Type (error 0x%lx)\n"),
WebViewGetLastError());
}

free(lpszValue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHeaders, GetFirstHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetHeaderCount Method

```
BOOL GetHeaderCount(
   DWORD dwHeaderType,
   LONG *lpnCount
);
```

The **GetHeaderCount** method returns the number of headers defined for the specified header type.

Parameters

dwHeaderType

An unsigned integer which specifies the type of header being checked. It may be one of the following values.

Name	Description
WEBVIEW_HEADERS_REQUEST	The number of custom headers submitted with a request.
WEBVIEW_HEADERS_RESPONSE	The number of headers returned by the server.

lpnCount

A pointer to an integer which will contain the number of headers when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method can be used to determine the number of custom request headers which have been set, or the number of headers returned by the server in response to a navigation. To enumerate the available response headers, use the **GetFirstHeader** and **GetNextHeader** methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearHeaders, GetFirstHeader, GetNextHeader, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetHttpStatus Method

```
INT GetHttpStatus();
```

The **GetHttpStatus** method returns the HTTP status code from the last completed navigation.

Parameters

None.

Return Value

If the method succeeds, the return value is a non-zero status code. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method returns the HTTP status code from the server response associated with the last completed navigation. These codes are defined by the HTTP standard and indicate the outcome of the request:

- **100-199**: Informational responses indicating the request is in progress.
- **200-299**: Successful responses, with 200 (OK) being the most common.
- **300-399**: Redirects, such as 301 (Moved Permanently) or 302 (Found).
- **400-499**: Client errors, such as 403 (Forbidden) or 404 (Not Found).
- **500-599**: Server errors, such as 500 (Internal Server Error) or 503 (Service Unavailable).

It is important to note that this value is only meaningful once navigation has completed. During a series of redirects, the status code may change multiple times; the value returned will be from the final response that completed navigation. A return value of `0` indicates the HTTP status code is unavailable. It may also return zero when the browser has navigated to a local file or when HTML content has been supplied directly using the **SetDocumentText** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetNavigationStatus, GetSecurityLevel, Navigate, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetLastError Method

```
DWORD GetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a function's return value indicates that an error has occurred. That is because some functions call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE or NULL. Those functions which call **SetLastError** when they succeed are noted on the method reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetErrorString, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetNavigationStatus Method

```
BOOL GetNavigationStatus(
   LPDWORD lpdwFlags,
   LPDWORD lpdwError
);
```

The **GetNavigationStatus** method returns the current navigation state.

Parameters

lpdwFlags

A pointer to an unsigned integer which may contain one of more status bitflags. This parameter may be NULL if the status flags are not required.

Name	Description
WEBVIEW_NAVIGATION_REDIRECTED	The navigation was the result of an HTTP redirect.
WEBVIEW_NAVIGATION_USER_INITIATED	Navigation began from a user gesture rather than being initiated by a script.
WEBVIEW_NAVIGATION_LOCAL_DATA	Content was served using local data rather than a full network fetch.

lpdwError

A pointer to a unsigned integer which will contain one of the following error codes. This parameter may be NULL if the error code is not required.

Name	Description
WEBVIEW_ERROR_NONE	No error.
WEBVIEW_ERROR_CERTIFICATE_COMMON_NAME	Certificate common name mismatch.
WEBVIEW_ERROR_CERTIFICATE_EXPIRED	Certificate is expired.
WEBVIEW_ERROR_INVALID_CLIENT_CERTIFICATE	Invalid or untrusted client certificate.
WEBVIEW_ERROR_CERTIFICATE_REVOKED	Certificate has been revoked.
WEBVIEW_ERROR_INVALID_CERTIFICATE	Certificate is invalid for another reason.
WEBVIEW_ERROR_SERVER_UNREACHABLE	Cannot reach the server/host.
WEBVIEW_ERROR_TIMEOUT	Operation timed out.
WEBVIEW_ERROR_INVALID_SERVER_RESPONSE	Server returned an invalid or unexpected response.
WEBVIEW_ERROR_CONNECTION_ABORTED	Connection aborted by the server or network.
WEBVIEW_ERROR_CONNECTION_RESET	Connection was reset.
WEBVIEW_ERROR_DISCONNECTED	Disconnected during the

	operation.
WEBVIEW_ERROR_CANNOT_CONNECT	Failed to establish a connection.
WEBVIEW_ERROR_HOST_NAME_NOT_RESOLVED	DNS resolution failed for the host.
WEBVIEW_ERROR_OPERATION_CANCELED	Operation was canceled by the user or host.
WEBVIEW_ERROR_REDIRECT_FAILED	Redirection failed.
WEBVIEW_ERROR_UNEXPECTED_ERROR	An unexpected error occurred.
WEBVIEW_ERROR_CREDENTIALS_REQUIRED	Authentication credentials are required.
WEBVIEW_ERROR_PROXY_AUTHENTICATION_REQUIRED	Proxy authentication is required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNavigationStatus** method provides detailed information about the current state of navigation for the specified WebView instance. It can be used to determine if a navigation is still in progress, whether it completed successfully, or if it was canceled or failed due to an error condition. The status flags returned in *lpdwFlags* can be combined to indicate additional context about the navigation, such as whether it was initiated by the user, if the content was served from local data rather than fetched over the network, or if the navigation resulted from an HTTP redirect.

Developers may find this method most useful in the following situations:

- After handling a `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event, to check if the navigation was successful.
- Before retrying or redirecting a failed request, to determine the specific cause of failure.
- When logging diagnostic information about connectivity or certificate errors.
- To distinguish between user-initiated cancellation, network timeouts, or security errors.
- To detect when content was provided from local data rather than fetched remotely.

The error code returned in *lpdwError* provides additional detail when a navigation fails or is blocked. These values cover a wide range of conditions, including invalid or expired certificates, connectivity issues, unreachable hosts, proxy authentication failures, and other network-related problems. If *lpdwError* is `WEBVIEW_ERROR_NONE`, the most recent navigation completed without error.

When navigation involves local files or HTML provided directly to the browser via **SetDocumentText**, the method will typically return `WEBVIEW_ERROR_NONE` and may set `WEBVIEW_NAVIGATION_LOCAL_DATA` in the status flags. This allows the application to detect that the content was not fetched over the network.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

CancelNavigation, GetHttpStatus, GetNavigationTime, Navigate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetNavigationTime Method

```
BOOL GetNavigationTime(
   LPDWORD lpdwElapsed,
   LPBOOL lpbNavigating
);
```

The **FunctionName** method returns the elapsed time for the last navigation.

Parameters

lpdwElapsed

A pointer to an unsigned integer which will contain the number of milliseconds elapsed since the current navigation has started.

lpbNavigating

A pointer to an integer which is updated when the method returns. It will contain a non-zero value if the browser is in the process of navigating to a new location. If the browser has completed navigation or the last navigation was canceled, this value will be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNavigationTime** method measures how long the most recent navigation has taken, reported in milliseconds. If a navigation is currently in progress, the elapsed time represents the duration since the navigation began. If the navigation has already completed, the elapsed time represents the total duration of that navigation. If no navigation has occurred, the elapsed time will be zero.

This method can be used for diagnostics, performance monitoring, or implementing application-level timeouts. For example, an application may track whether a page is taking unusually long to load and decide to cancel the navigation, notify the user, or retry the request. Combined with **GetNavigationStatus**, it provides a way to determine both the state and timing of navigation operations.

Typical usage scenarios for this method include:

- Measuring page load times for performance profiling or telemetry.
- Implementing a custom navigation timeout by checking the elapsed time in a periodic timer.
- Providing user feedback, such as displaying how long a page has been loading.
- Logging navigation duration along with success or failure status for diagnostic purposes.

The *lpbNavigating* parameter indicates whether a navigation is currently in progress when the method is called. This allows developers to distinguish between a completed navigation (where the elapsed time is final) and an in-progress navigation (where the elapsed time is still increasing).

It is important to note that the reported time is based on the browser's internal navigation start and completion timestamps. It reflects the overall duration of the navigation process as observed by the browser, not fine-grained network-level timings such as DNS lookup, TCP connection, or TLS handshake. For most application scenarios this high-level timing is sufficient to evaluate page load performance and detect unusual delays.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetNavigationStatus, Navigate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetNextHeader Method

```
BOOL GetNextHeader(
   LPTSTR lpszHeaderName,
   INT nMaxNameLen,
   LPTSTR lpszHeaderValue,
   INT nMaxValueLen
);
```

The **GetNextHeader** method returns the next name/value pair in a response header enumeration.

Parameters

lpszHeaderName

A string buffer which will contain the name of the first header returned by the server. If this parameter is NULL it will be ignored.

nMaxNameLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderName* string buffer, including the terminating null. If the *lpszHeaderName* parameter is NULL this value should be zero.

lpszHeaderValue

A string buffer which will contain the value of the first header returned by the server. If this parameter is NULL, it will be ignored.

nMaxValueLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderValue* string buffer, including the terminating null. If the *lpszHeaderValue* parameter is NULL this value should be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method continues the enumeration that was started when the **GetFirstHeader** method was called. Repeatedly call **GetNextHeader** until it returns zero (FALSE) indicating there are no more header values to return. The **GetLastError** function can be used to distinguish between an actual error condition and the end of the enumeration. If the last error code is `ST_ERROR_END_OF_DATA` then all response headers have been enumerated.

If the provided buffers are too small, the header names and values will be truncated to the available space specified by the maximum length parameters. In most cases, a header name is not going to exceed 256 characters and a header value will normally not exceed 5120 characters. Most header values are relatively short, but some headers such as `Set-Cookie` can return much longer values.

This method cannot be called while the browser is in the process of navigating to a new location. It will also fail if the current document is a local file or if the document text was set dynamically using **SetDocumentText**. There are no response headers available in a local context.

Example

```
TCHAR szName[WEBVIEW_MAX_HEADER_NAME];
TCHAR szValue[WEBVIEW_MAX_HEADER_VALUE];
```

```
if (pWebView->GetFirstHeader(szName, _countof(szName),
                             szValue, _countof(szValue)))
{
    do
    {
        _tprintf(_T("%s: %s\n"), szName, szValue);
    }
    while (pWebView->GetNextHeader(szName, _countof(szName),
                                   szValue, _countof(szValue)));
}
else
{
    DWORD dwError = pWebView->GetLastError();
    _tprintf(_T("No headers available (error 0x%lx)\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHeaders, GetFirstHeader, GetHeader, GetHeaderCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetOption Method

```
BOOL GetOption(
   DWORD dwOption
   LPBOOL lpbEnabled
);
```

The **GetOption** method checks whether a specific option flag is enabled.

Parameters

dwOption

An unsigned integer which specifies the option to be checked. Refer to the Options documentation for the list of available options. Only a single option may be specified; you cannot combine multiple option bitflags when calling this method.

lpbEnabled

A pointer to an integer which will be updated when the method returns. If the value is non-zero, the option has been enabled. If the value is zero, the option has not been enabled.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetOption** method is used to determine if a specific option has been enabled for the browser instance associated with the specified host window. Each option corresponds to an individual feature or behavior that can be enabled or disabled at creation time or later using **SetOption**.

This method only supports checking a single option value at a time. The *dwOption* parameter must specify exactly one option bitflag, such as `WEBVIEW_OPTION_ENABLE_SCRIPTING` or `WEBVIEW_OPTION_ENABLE_CONTEXT_MENU`. It is not valid to combine multiple bitflags with a bitwise OR. If you need to retrieve the complete set of enabled options, call **GetAllOptions**.

Note that not all options may be applicable after a window has been created. Some options only take effect at creation time and cannot be changed dynamically. For example, options related to profile isolation or privacy modes must be specified before the control is initialized.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, GetAllOptions, SetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetSafeHwnd Method

```
HWND GetSafeHwnd();
```

The **GetSafeHwnd** method returns the handle to the host window.

Parameters

None.

Return Value

If the method succeeds, the return value is a window handle. If the method fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

This method is similar to **CWnd::GetSafeHwnd** in that it returns NULL if the class is not attached to a browser instance. This method would typically be used to retrieve the window handle created using the **Create** method. Note that this is not the handle to the child window used by the WebView2 browser, it is the handle to the window hosting the browser. To get the browser child window, use the **GetChildWindow** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetChildWindow, Redraw, SetBounds, SetFocus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetSecurityInfo Method

```
BOOL GetSecurityInfo(
   WEBVIEW_SECURITY_INFO *lpSecurityInfo
);
```

The **GetSecurityInfo** method returns certificate and host security details for the last navigation.

Parameters

lpSecurityInfo

A pointer to a `WEBVIEW_SECURITY_INFO` structure which will contain information about the specified download when the method returns. The ***dwSize*** member of this structure must be initialized to the size of the structure prior to being passed to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetSecurityInfo** method retrieves details about the security state of the current page and populates a `WEBVIEW_SECURITY_INFO` structure. This includes the negotiated TLS protocol, cipher suite, certificate subject and issuer, and the validity period of the certificate. The method also reports an overall security level (***dwLevel***) and condition-specific flags (***dwFlags***) that identify issues such as expired certificates, mixed content, or weak cryptographic algorithms.

The information returned reflects the most recent navigation state. During navigation, these structure values may be provisional. For example, a page may initially appear secure but later be downgraded to a warning state if mixed content is detected. Applications should consider the values as advisory until a `WEBVIEW_EVENT_NAVIGATION_COMPLETED` notification confirms the final state.

The security level (***dwLevel***) is intended for user interface decisions, such as whether to display a lock icon. The flags (***dwFlags***) provide more granular context and may include multiple indicators at once. For instance, a site may be considered secure overall but still include a `WEBVIEW_SECURITY_FLAG_CERT_EXPIRED` flag if the certificate is past its validity period. Because the browser can revise its assessment as resources are loaded, applications that display security indicators should refresh their state after navigation completes. Do not assume that an initially secure status will remain unchanged for the life of the page. Conditions such as mixed content or blocked subresources may alter the security level after the main document is retrieved.

For plain HTTP or local content, the ***szProtocol*** and ***szCipher*** members will be empty, ***dwFlags*** will typically include `WEBVIEW_SECURITY_FLAG_UNENCRYPTED`, and the security level will be `WEBVIEW_SECURITY_LEVEL_INSECURE` or `WEBVIEW_SECURITY_LEVEL_NEUTRAL`. For HTTPS with a valid certificate, the protocol and cipher suite are reported, the subject and issuer strings identify the certificate, and the security level will be `WEBVIEW_SECURITY_LEVEL_SECURE`.

The subject and issuer members are provided for display or logging purposes only; certificate parsing and validation are handled internally by the browser engine. Likewise, the validity dates (***ftValidFrom*** and ***ftValidTo***) are exposed for simple checks but do not require applications to perform their own certificate validation.

Before calling this method, the ***dwSize*** member of the structure must be initialized to the size of the structure. This ensures compatibility with future versions of the library if additional fields are introduced. If the size does not match, the function will fail or only initialize the subset of fields it

recognizes.

Both ANSI (A) and Unicode (W) variants are available. When compiling with UNICODE defined, string members are UTF-16 encoded; otherwise, they are UTF-8 encoded. The **GetSecurityInfo** macro resolves to the appropriate version based on the project's character set.

If you are only interested in determining the current security level of the connection you can use the **GetSecurityLevel** function.

Example

```
// Map security level to text
static LPCTSTR SecLevelToText(DWORD level)
{
    switch (level)
    {
    case WEBVIEW_SECURITY_LEVEL_SECURE: return _T("SECURE");
    case WEBVIEW_SECURITY_LEVEL_WARNING: return _T("WARNING");
    case WEBVIEW_SECURITY_LEVEL_INSECURE: return _T("INSECURE");
    case WEBVIEW_SECURITY_LEVEL_NEUTRAL: return _T("NEUTRAL");
    default: return _T("UNKNOWN");
    }
}

// Print a FILETIME value as YYYY-MM-DD HH:MM:SS
static void PrintFileTime(LPCTSTR label, const FILETIME* pft)
{
    if (!pft || (pft->dwLowDateTime == 0 && pft->dwHighDateTime == 0))
    {
        _tprintf(_T("%s: (not available)\n"), label);
        return;
    }

    FILETIME lft = {0};
    SYSTEMTIME st = {0};

    if (FileTimeToLocalFileTime(pft, &lft) && FileTimeToSystemTime(&lft, &st))
    {
        _tprintf(_T("%s: %04u-%02u-%02u %02u:%02u:%02u\n"),
                 label,
                 (unsigned)st.wYear,
                 (unsigned)st.wMonth,
                 (unsigned)st.wDay,
                 (unsigned)st.wHour,
                 (unsigned)st.wMinute,
                 (unsigned)st.wSecond);
    }
    else
    {
        _tprintf(_T("%s: (conversion failed)\n"), label);
    }
}

// Build and print a readable flag list
static void PrintSecurityFlags(DWORD dwFlags)
{
    if (dwFlags == WEBVIEW_SECURITY_FLAG_NONE)
    {
        _tprintf(_T("Flags: NONE (0x%08lX)\n"), dwFlags);
```

```
        return;
    }

    _tprintf(_T("Flags: 0x%08lX"), dwFlags);

    if (dwFlags & WEBVIEW_SECURITY_FLAG_UNENCRYPTED) _tprintf(_T(" |
UNENCRYPTED"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_INSECURE) _tprintf(_T(" | INSECURE"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_CERT_WEAK) _tprintf(_T(" | CERT_WEAK"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_CERT_ERROR) _tprintf(_T(" | CERT_ERROR"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_CERT_EXPIRED) _tprintf(_T(" |
CERT_EXPIRED"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_UNSAFE) _tprintf(_T(" | UNSAFE"));

    _tprintf(_T("\n"));
}

// Query and display the current security info
static void ShowSecurityInfo(CWebView *pWebView)
{
    WEBVIEW_SECURITY_INFO secInfo = { 0 };
    secInfo.dwSize = sizeof(secInfo);

    if (!pWebView->GetSecurityInfo(&secInfo))
    {
        _tprintf(_T("GetSecurityInfo failed (Error 0x%lx)\n"), pWebView-
>GetLastError());
        return;
    }

    _tprintf(_T("Security Level: %s\n"), SecLevelToText(secInfo.dwLevel));
    PrintSecurityFlags(secInfo.dwFlags);

    if (secInfo.szProtocol[0])
        _tprintf(_T("Protocol: %s\n"), secInfo.szProtocol);
    else
        _tprintf(_T("Protocol: (none)\n"));

    if (secInfo.szCipher[0])
        _tprintf(_T("Cipher Suite: %s\n"), secInfo.szCipher);
    else
        _tprintf(_T("Cipher Suite: (none)\n"));

    if (secInfo.szSubject[0])
        _tprintf(_T("Certificate Subject: %s\n"), secInfo.szSubject);
    else
        _tprintf(_T("Certificate Subject: (not available)\n"));

    if (secInfo.szIssuer[0])
        _tprintf(_T("Certificate Issuer: %s\n"), secInfo.szIssuer);
    else
        _tprintf(_T("Certificate Issuer: (not available)\n"));

    PrintFileTime(_T("Valid From"), &secInfo.ftValidFrom);
    PrintFileTime(_T("Valid To "), &secInfo.ftValidTo);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetSecurityLevel, IsSecure, WEBVIEW_SECURITY_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetSecurityLevel Method

```
BOOL GetSecurityLevel(
   LPDWORD lpdwLevel
);
```

The **GetSecurityLevel** method returns the evaluated security level for the last navigation.

Parameters

lpdwLevel

A pointer to an unsigned integer which will contain the security level when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetSecurityLevel** method provides a simplified way to obtain the current security state of the page displayed in the browser. This value is an overall assessment of the page's security status and is primarily intended for use in user interface logic, such as whether to display a lock icon or warning indicator. The security level does not provide detailed diagnostic information; if your application requires additional context such as certificate details, protocol version, or specific security flags, you should call **WebViewGetSecurityInfo** instead.

The possible values returned in the *lpdwLevel* parameter correspond to the security level constants:

- `WEBVIEW_SECURITY_LEVEL_UNKNOWN`: The security state could not be determined.
- `WEBVIEW_SECURITY_LEVEL_INSECURE`: The page was retrieved over plain HTTP or the certificate is invalid.
- `WEBVIEW_SECURITY_LEVEL_WARNING`: The page uses weak encryption or contains mixed content.
- `WEBVIEW_SECURITY_LEVEL_NEUTRAL`: Local content or the browser is in a transitional state.
- `WEBVIEW_SECURITY_LEVEL_SECURE`: The page was retrieved using HTTPS with a valid certificate.

Because the browser engine may revise the security assessment as additional resources are loaded, the level can change over the course of a navigation. For example, a page may initially appear secure, but once mixed content is detected the level will be downgraded to `WEBVIEW_SECURITY_LEVEL_WARNING.` Applications should refresh their indicators when they receive a `WEBVIEW_EVENT_SECURITY_CHANGED` notification or after navigation has completed.

The level values are ordered, with higher values indicating stronger security. Applications should avoid upgrading an observed level within the same navigation, since subsequent checks may legitimately lower the assessment.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetSecurityInfo, IsSecure

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for the navigation to complete.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. A value of zero specifies no timeout period has been specified.

Remarks

The **GetTimeout** method retrieves the current navigation timeout value, expressed in seconds. This value determines how long the library will wait for a navigation request to complete before it is considered to have failed due to a timeout condition.

If the timeout value is set to zero, the control uses the built-in timeouts defined by the underlying WebView2 browser engine. These internal timeouts are adaptive and depend on network conditions, DNS resolution, and connection establishment. In practice, this allows the browser to determine how long to wait before treating a navigation as unresponsive. For most applications, leaving the timeout set to zero provides behavior consistent with the Microsoft Edge browser.

If a non-zero timeout value has been set, navigation will be canceled if the specified number of seconds elapses without the page load completing. This provides a deterministic limit on how long a navigation attempt is allowed to continue before the operation fails. However, specifying a very long timeout value does not guarantee that navigation will continue for that entire duration, since the WebView2 engine may still enforce its own internal limits and terminate the request earlier.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetUserAgent Method

```
BOOL GetUserAgent(
   LPTSTR lpszUserAgent,
   INT nMaxLength,
);
```

The **GetUserAgent** method returns the current User-Agent value for this instance.

Parameters

lpszUserAgent

A null-terminated string buffer which will contain the User-Agent string for the current session. If this parameter is NULL, the string will not be copied and the method will return the length of the string, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the *lpszUserAgent* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters in the User-Agent string, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetUserAgent** method retrieves the current User-Agent string used by this browser instance. This value is the exact string sent in the User-Agent header of outgoing HTTP requests and is determined by the underlying Microsoft Edge WebView2 runtime unless explicitly changed with **SetUserAgent**.

The returned value can be used for informational purposes, such as logging the client identity or displaying diagnostic information to the user. It is also useful if your application needs to capture and preserve the default User-Agent string before setting a custom one.

If the *lpszUserAgent* parameter is `NULL`, the method returns the length of the string in characters. This allows the caller to allocate a buffer of the appropriate size before retrieving the value. When a valid buffer is provided, the string copied to it will always be null-terminated.

The User-Agent value applies to the entire lifetime of the browser instance. Once it has been changed, all subsequent navigations will use the new string until explicitly reset or the instance is destroyed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, SetHeader, SetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetUserDataFolder Method

```
INT GetUserDataFolder(
   LPTSTR lpszDataFolder,
   INT nMaxLength,
);
```

The **GetUserDataFolder** method returns the full path to the user data folder for the current session.

Parameters

lpszDataFolder

A null-terminated string buffer which will contain the path to user data folder for the session. If this parameter is NULL, the string will not be copied and the method will return the length of the path, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the ***lpszDataFolder*** parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters in the User-Agent string, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **User Data Folder** (UDF) is the WebView2 profile directory that Chromium uses to persist browsing state for this instance. It contains the browser cache, cookies, localStorage/IndexedDB data, service worker state, visited link history, site permissions, and other profile data. Each ***distinct*** folder path corresponds to a distinct browser profile. Multiple WebView2 instances that point to the same folder will share the same profile (and therefore share cookies, storage, and authentication state). Using different folder paths isolates profiles.

The user data folder is chosen when the WebView instance is created or attached via **Attach** or **Create** and cannot be changed afterwards. If your application needs to switch profiles (for example, to use a different folder), you must destroy the current instance and create a new one with the desired folder.

If your application does not specify a folder or uses an empty string as the folder path, the library creates a unique per-session folder under the current user's local profile. This is an ***ephemeral profile*** that is not intended to be reused across browser instances. By default, this temporary folder is removed when the instance is detached; if the process terminates unexpectedly, the folder may remain and can be safely deleted later.

In contrast, specifying an explicit, persistent folder (for example, a subdirectory under `%LocalAppData%\WebView\YourApp`) gives you a shared profile that persists across sessions and can be reused by multiple WebView2 instances to enable single sign-on, shared cookies, and faster subsequent navigations due to caching. Use this approach when you want consistent, long-lived browser state.

The reported folder path is always an absolute, normalized local path. Environment variables in the path you supply at creation time are expanded; forward slashes are normalized to backslashes and relative names are resolved beneath the current user's local application data folder. The library rejects invalid or unsafe locations (e.g.: device names, system directories, UNC or extended path prefixes, or

root drive targets) and will create intermediate directories as needed for valid paths. For multithreaded applications, you should use explicit absolute paths rather than relying on the process current directory.

Important: The user data folder is managed internally by the the WebView2 runtime, and your application should never attempt to modify, move, or delete any of the files it contains while a browser instance is active. The contents are in use by the rendering engine and interfering with them can result in data corruption or unexpected behavior. If you need to remove or relocate a profile, ensure that all browser instances using that folder have been completely closed. If you want to delete certain data, such as cookies or the browser history, use the methods provided specifically for that purpose.

You can call **GetUserDataFolder** at any time after the instance has been created to discover which profile directory is in use. Passing *lpszDataFolder* as `NULL` returns the required buffer length so you can allocate a buffer of the appropriate size; the returned length does not include the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Attach, Create

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetWebMessage Method

```
BOOL GetWebMessage(
   LPTSTR lpszMessage,
   INT nMaxLength
);
```

The **GetWebMessage** method retrieves the next pending message sent from the current document.

Parameters

lpszMessage

A null-terminated string buffer which will contain the message sent from the current document. If this parameter is NULL, the message will not be copied and the method will return the length of the message, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the *lpszMessage* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters in the message, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetWebMessage** method retrieves the most recent message sent from script code running in the browser. This is typically used to enable interaction between the web content displayed in the browser and the host application. For example, a page might notify the application when a user completes a form, clicks a button, or when some script logic produces a result that the native application needs to act upon. This provides a simple and secure channel for exchanging data between the web page and the host, without requiring the page to reload or navigate to a new URL.

Messages are sent from JavaScript using the following method:

```
window.chrome.webview.postMessage(value);
```

When a script in the browser sends a message, the `WEBVIEW_EVENT_MESSAGE_RECEIVED` event will fire and your event handler should call this method to retrieve the value posted by the script. The `value` parameter can be any JSON-serializable object or a simple string. The format of the message depends on the type of value that was posted:

- If the message was a structured JavaScript object or array, it will be returned as a JSON object string using a format like `{"key":"value"}`.
- If the message was a string, it will be returned as a JSON-formatted string with surrounding quotes and any special characters will be escaped, such as `"Hello\nWorld"`.

Applications that need to process these messages should parse the result as JSON to properly distinguish between plain strings and structured data.

Important: Message passing is disabled by default. To enable this functionality, the browser must be created or attached with the `WEBVIEW_OPTION_ENABLE_MESSAGES` flag. If this option is not specified, no messages will be received and this method will not return any content.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Attach, Create, GetDocumentText, GetWebResult, PostWebMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetWebResult Method

```
BOOL GetWebResult(
   LPTSTR lpszResult,
   INT nMaxLength
);
```

The **GetWebResult** method returns the result from the most recent completed script execution.

Parameters

lpszResult

A null-terminated string buffer which will contain the result from the script that was previously executed. If this parameter is NULL, the result will not be copied and the method will return the length of the string, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters in the result string, not including the terminating null character. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetWebResult** method retrieves the result of the most recent JavaScript expression executed in the browser using either the **AsyncExecute** or **Execute** methods. The result is provided as a JSON-formatted string, which may be either a quoted string with escaped characters or a JSON object, depending on the value returned by the script. Applications that expect structured data should parse the returned string using a JSON parser.

Only the result from the most recently executed script is retained. Executing another script will overwrite the previous result, even if the value has not yet been retrieved. If your application needs to preserve earlier results, it should copy the returned string into its own buffer for later use.

When using **AsyncExecute**, the result is available once the `WEBVIEW_EVENT_SCRIPT_COMPLETED` event has been raised. Attempting to call **GetWebResult** before the event is delivered will return an empty string. When using **Execute**, the script is executed synchronously and the result is immediately available.

The maximum length of the result string is limited by the size of the buffer provided. If the *lpszResult* parameter is NULL, the function will return the required length of the result string in characters, not including the terminating null, which can be used to allocate a buffer of the appropriate size.

Example

```
// Execute a simple script that returns a string
INT nLength = pWebView->Execute(_T("document.title"));

if (nLength > 0)
{
    // Allocate a buffer large enough to hold the result
    LPTSTR lpszResult = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (lpszResult)
```

```
    {
        // Retrieve the result of the script execution
        if (pWebView->GetWebResult(lpszResult, nLength + 1))
        {
            _tprintf(_T("Page title is: %s\n"), lpszResult);
        }
        free(lpszResult);
    }
}
else
{
    _tprintf(_T("Script execution failed\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AsyncExecute, Execute, GetWebMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GetZoomLevel Method

```
BOOL GetZoomLevel(
   INT *lpnPercent
);
```

The **GetZoomLevel** method returns the current zoom level as a percentage.

Parameters

lpnPercent

A pointer to an integer which will contain the current zoom level as a percentage value when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetZoomLevel** method returns the current zoom level of the browser window as an integer percentage value. A zoom level of **100** indicates that the page is being displayed at its normal size without any scaling applied. Values greater than **100** mean that the content is being magnified, and values less than **100** mean that the content is being reduced.

The zoom level is constrained by the limits defined for the control. By default, the minimum zoom level is **25** (25%) and the maximum zoom level is **400** (400%). Attempts to set values outside of this range using **SetZoomLevel** will be clamped to the supported range.

This method can be used to determine the current scaling when an application wants to preserve a user's preferred zoom level across sessions, or to query whether the content is being displayed at the default size. If the method fails, the contents of the ***lpnPercent*** parameter are undefined and should not be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

SetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GoBack Method

```
BOOL GoBack();
```

The **GoBack** method navigates backwards through the browser history.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GoBack** method navigates the browser to the previous page in its session history, if one is available. This is equivalent to selecting the Back button in a standard web browser. If there is no previous entry in the history list, the method will succeed but no navigation will occur. The availability of backward navigation can be determined by handling the `WEBVIEW_EVENT_HISTORY_CHANGED` event and checking for the `WEBVIEW_MOVE_BACKWARD_ENABLED` flag.

Calling **ClearHistory** will remove the session history for the browser instance. After the history has been cleared, calling **GoBack** will have no effect until new pages have been visited.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearHistory, GoForward, Navigate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::GoForward Method

```
BOOL GoForward();
```

The **GoForward** method navigates forwards through the browser history.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GoForward** method navigates the browser to the next page in its session history, if one is available. This is equivalent to selecting the Forward button in a standard web browser.

If there is no forward entry in the history list, the method will succeed but no navigation will occur. The availability of forward navigation can be determined by handling the `WEBVIEW_EVENT_HISTORY_CHANGED` event and checking for the `WEBVIEW_MOVE_FORWARD_ENABLED` flag.

Calling **ClearHistory** will remove the session history for the browser instance. After the history has been cleared, calling **GoForward** will have no effect until new pages have been visited.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearHistory, GoBack, Navigate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Hide Method

```
BOOL Hide();
```

The **Hide** method hides the browser window.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Hide** method changes the visibility state of the browser control. When the browser was created as a top-level window using **Create**, this method internally calls the standard **ShowWindow** API to hide the entire browser window. The window will be restored with the **Show** method.

When the browser is embedded in an existing host window using **Attach**, the method does not change the visibility of the parent window. Instead, it sends an internal message to the control to hide the child window which contains the WebView2 browser. This enables the host application to change the visibility of the browser control independently of other controls in the same window.

This method does not destroy the browser instance or unload its current content. Hiding the browser simply makes it invisible; it can be shown again without requiring a reload of the current document. The current visibility state can be queried using the **IsVisible** method.

Hiding the browser window can also provide performance benefits in scenarios where the application does not need to display the page contents for a period of time. While background tasks such as script execution and network activity continue as normal, the browser will not spend resources on rendering or updating the visual output when the control is not visible. This can reduce CPU and GPU usage, particularly when the page is dynamic or graphically intensive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetFocus, IsVisible, Redraw, SetBounds, SetFocus, Show

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::InEvent Method

```
BOOL InEvent();
```

The **InEvent** method determines if the current thread is executing code within an event handler.

Parameters

None.

Return Value

If the current thread is executing code within the context of an event handler, the return value is non-zero; otherwise the return value is zero.

Remarks

The **InEvent** method checks a thread-local flag that is set whenever your code is executing inside an event handler. This allows an application to determine if the current call stack is the result of an event callback (for example, a navigation completed event or a message received event) rather than code executing on its own.

Knowing whether you are inside a event handler can be useful for several reasons:

- **Reentrancy detection**. Certain API functions should not be called while an event is still being processed, since this can cause re-entrant behavior or unexpected state changes in the underlying WebView2 control. For example, it is not permitted to attempt another navigation or reset the browser state while inside a navigation event handler.
- **Thread marshalling**. Event handlers are typically invoked on the UI thread associated with the WebView instance. By checking this method, the application can decide whether it needs to marshal an operation to the browser's thread or whether it is already executing in the correct context.
- **Application logic**. In some cases you may want to avoid triggering further navigation, script execution, or message posting in direct response to a particular event. This method lets you distinguish between code running in an event handler versus code running elsewhere in your application.

The return value from this method is only valid for the calling thread. Each thread maintains its own state, so a worker thread will always return FALSE unless it is actively executing a event callback on that thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::IsAttached Method

```
BOOL IsAttached();
```

The **IsAttached** method determines if the class instance is attached to a window.

Parameters

None.

Return Value

If the class instance is attached to a host window, the return value is non-zero. If class is not attached to a window, the return value is zero.

Remarks

The **IsAttached** method checks whether the class instance has been attached to a window with the **Attach** method, or the class has been used to create a new top-level browser window with the **Create** method. To get the window handle to the host window, use the **GetSafeHwnd** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, GetSafeHwnd

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

When the class is initialized, it will also attempt to load the WebView2 runtime. The calling thread must be in a single-threaded apartment (STA). During initialization the class calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), initialization will fail. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

The **RuntimeAvailable** method can be used to determine if the Microsoft Edge WebView2 runtime is installed and meets the minimum required version. This can be useful during application startup to provide a clear error message or guidance to the user if the runtime is missing or out of date, rather than attempting to initialize the control and handling the failure afterward. This is a static method which can be called without creating an instance of the **CWebView** class.

On modern versions of Windows, the WebView2 runtime is included as part of the operating system and is serviced through Windows Update. However, on older or unsupported versions of Windows the runtime may not be present by default and must be installed separately. In these cases, your application needs to detect the missing runtime and prompt the user to install it.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

CWebView, IsAttached, RuntimeAvailable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::IsSecure Method

```
BOOL IsSecure();
```

The **IsSecure** method determines if the current page was retrieved using a secure connection.

Parameters

None.

Return Value

If the connection was secure, the return value is non-zero. If the connection was not secure or there were problems (such as an expired certificate), the return value is zero.

Remarks

The **IsSecure** method checks whether the current page was retrieved using a secure connection. Because the security state of the session changes several times as a page is loaded, this value should only be considered authoritative after navigation has completed. It is possible for the method to initially return **TRUE** as the page is loading, but return **FALSE** when navigation completes if the page contains mixed content.

This method only returns **TRUE** if the document and all its assets were retrieved using a secure connection. The security certificate must be valid and the connection cannot use weak encryption or deprecated cipher suites. If a secure connection was used but there were problems detected with the certificate or security protocol, this method will return **FALSE**. For more information about the connection, use the **GetSecurityInfo** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetSecurityInfo, GetSecurityLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::IsVisible Method

```
BOOL IsVisible();
```

The **IsVisible** method determines if the browser window is currently visible.

Parameters

None.

Return Value

If the browser window is visible, the return value is non-zero. If the window is not visible or has been closed, the return value is zero.

Remarks

The **IsVisible** method checks whether the browser window is currently visible. A return value of **TRUE** means that the browser window is being displayed. A return value of **FALSE** indicates that the window has been hidden using the **Hide** method or the window was closed.

This method only reports on the visibility state that was explicitly set through the API. It does not reflect other factors that might affect whether the browser content is actually visible to the user, such as the host window being minimized, overlapped by another window, or located off screen.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetFocus, Hide, Redraw, SetBounds, SetFocus, Show

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::KeyEventHandled Method

```
BOOL KeyEventHandled(
   BOOL bHandled
);
```

The **KeyEventHandled** method sets an accelerator key event as handled.

Parameters

bHandled

An integer which specifies if the accelerator key event was handled by the application. A non-zero value indicates the application has handled the accelerator key event and the WebView2 control should not process the event. A value of zero specifies that the browser should handle the accelerator key using it's normal, default behavior.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **KeyEventHandled** method is used in conjunction with the **WebViewEventProc** callback when the application receives a `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS` event. By calling this method and specifying that the key event has been handled, the application can prevent the browser from processing the key itself.

This mechanism is useful when you want to disable or override the default behavior of certain keys within the embedded browser. For example, you may wish to prevent the user from pressing **F12** to open the developer tools, block function keys that have meaning in your application, or intercept combinations such as **Ctrl+P** or **Ctrl+S** to substitute your own print or save functionality.

If *bHandled* is non-zero, the event is considered handled by the host application and WebView2 will not process the key. If *bHandled* is zero, the key is passed through and processed normally according to the browser's default behavior. This method should typically be called during the event handler for `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS`.

Note that this method only affects accelerator key events when the `WEBVIEW_OPTION_ENABLE_ACCELERATORS` option has been specified when creating or attaching the WebView. It does not apply to ordinary text input within form fields or content editable regions in the page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Navigate Method

```
BOOL Navigate(
   DWORD dwFlags,
   LPCTSTR lpszUrl
);
```

The **Navigate** method navigates to the specified URL or local file path.

Parameters

dwFlags

An unsigned integer which specifies additional options for how the navigation will be performed.

Value	Description
WEBVIEW_NAVIGATE_DEFAULT	Use the default navigation behavior. The specified URL or local file path is normalized and used without additional restrictions.
WEBVIEW_NAVIGATE_SAFE_URL	Perform additional safety checks on the input before navigation. This will reject unsafe or malformed URLs, filter out potentially harmful schemes, and block traversal sequences (such as `../../`) in local paths. This option is recommended when the URL or path is provided by external or untrusted input.

lpszUrl

A null-terminated string which specifies the URL or local file to navigate to. Any leading and trailing whitespace will be removed and the URL will be normalized to ensure that it specifies a valid location. If the URL only specifies a domain name (such as `"example.com"`) the browser will default to using a secure connection. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Navigate** method accepts both standard web URLs such as `https://www.example.com/` and local file paths which are automatically normalized and converted into `file://` URLs. This allows an application to load either remote web content or content from the local filesystem.

When a local file path is specified, the method validates the path to ensure that it refers to a valid, absolute file system location and is not under restricted system folders (such as the Windows or Program Files directories). Relative paths are resolved against the current working directory, but it is recommended that applications provide absolute paths to avoid ambiguity in multithreaded or service-hosted environments.

File names and paths are normalized before navigation. Illegal or reserved device names (such as `CON`, `AUX`, or `NUL`) and paths containing invalid characters will be rejected. UNC paths and extended-length paths using the `\\?\` prefix are not supported. If the path points to a valid file, it will be loaded into the WebView using the appropriate `file://` URL form.

If the *dwFlags* parameter specifies `WEBVIEW_NAVIGATE_SAFE_URL`, the URL or path will undergo

additional checks to ensure that it does not contain unsafe schemes, fragments, or traversal sequences. This is useful when the application accepts untrusted input that may be used for navigation.

Note that navigation to local files is supported, but applications should exercise caution when mixing remote and local content, as this may have cross-origin security implications. If your application needs to load dynamic HTML content, consider using **SetDocumentText** instead.

This method only supports navigation to standard web and local resources. Valid URL schemes are limited to `http://`, `https://`, `file://`, and `about:blank`. Other protocols or custom application schemes are not supported and will cause the method to fail. This includes the `edge://settings/profile` page or any other internal configuration settings for the Edge browser. If your application needs to handle such protocols, you should implement a custom handler to intercept those links and handle them externally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCurrentUrl, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::NormalizeUrl Method

```
INT NormalizeUrl(
   LPCTSTR lpszUrl,
   DWORD dwFlags,
   LPTSTR lpszNormalized,
   INT nMaxLength
);

INT NormalizeUrl(
   LPCTSTR lpszUrl,
   DWORD dwFlags,
   CString& strNormalized
);
```

The **NormalizeUrl** method parses and normalizes a URL into a safe, navigable form.

Parameters

lpszUrl

A null-terminated string which specifies the URL to be normalized. This parameter cannot be NULL or an empty string.

dwFlags

Specifies one or more bit flags that control how the URL is normalized. A value of zero specifies that only standard normalization will be performed, such as removing any leading and trailing whitespace and ensuring the URL does not contain any illegal characters. Multiple flags may be combined using a bitwise or operator.

Value	Description
URL_FLAG_DEFAULT_HTTPS	When no URL scheme is provided and no scheme is otherwise required, select **`https://`** by default. If an explicit port is present, port 443 selects **`https`** and any other port selects **`http`**. Explicit input schemes are preserved.
URL_FLAG_REQUIRE_HTTP	When no scheme is provided, force **`http://`**. If the input already specifies a scheme, it is preserved.
URL_FLAG_REQUIRE_HTTPS	When no scheme is provided, force **`https://`**. If the input already specifies a scheme, it is preserved.
URL_FLAG_REJECT_LOCALHOST	Reject URLs that specify **`localhost`** as the host name.
URL_FLAG_REJECT_IPV4	Reject URLs that specify a numeric IPv4 address (for example, **`192.0.2.1`**) as the host.
URL_FLAG_REJECT_IPV6	Reject URLs that specify a bracketed IPv6 address (for example, **`[2001:db8::1]`**) as the host.
URL_FLAG_ASCII_HOSTNAME	Host names are checked to ensure that they only contain legal ASCII characters. Unicode host names are converted to ASCII (punycode) before validation, and the host name is normalized to lowercase.
URL_FLAG_ESCAPE_PATH	Percent-encode the path component as needed. Non-ASCII characters are encoded as UTF-8 and then percent-encoded

	per byte. Valid **%HH** sequences are preserved and upper-cased. The path is also normalized by removing duplicate slashes and collapsing **.** and **..** segments of the path.
URL_FLAG_ESCAPE_QUERY	Percent-encode the query and fragment tail as needed. The first **?** is treated as the start of the tail and characters commonly used in queries (such as **&**, **=**, **+**, **;**, **#**) are permitted unescaped. Non-ASCII characters are encoded as UTF-8 and then percent-encoded.
URL_FLAG_VALIDATE_UTF8	Applicable to projects using a multi-byte character set. Require the URL to use UTF-8 encoding and fail if the method encounters an invalid Unicode character. When this flag is not specified, and the URL does not use UTF-8 encoding, the method attempts to normalize the URL using the active code page.
URL_FLAG_REJECT_LOCAL_FILES	Reject absolute local and UNC paths and existing **file://** URLs. When specified, UNC paths are also rejected and unsafe file extensions are not allowed.
URL_FLAG_REJECT_UNC_PATHS	When local file normalization is permitted, reject UNC paths (for example, **\\server\share**). Files paths which use a mapped drive to a network share will be allowed.
URL_FLAG_ALLOW_UNSAFE_FILES	When converting local/UNC paths to **file://** URLs or normalizing existing **file://** URLs, permit any file extension. If this option is not specified, only a common set of safe, renderable file types are allowed. This includes HTML files, text files and common image formats.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the method returns. This parameter may be NULL, in which case the method returns the number of characters required to store the normalized URL. If a buffer is provided, it must be large enough to contain the entire URL. An alternate version of this method accepts a **CString** which will contain the normalized URL when the method returns.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. If this parameter is zero, the method will return the number of characters required to store the normalized URL and no characters will be copied into the buffer.

Return Value

An integer which specifies the length of the normalized URL, not including the terminating null character. A return value of zero indicates the URL could not be normalized, either due to an invalid URL or the output buffer not being large enough to store the complete URL. Call the **GetLastError** method for more information about the cause of the failure.

Remarks

The **NormalizeUrl** method parses and normalizes a candidate URL without navigating. In most cases you do not need to call this method directly because the **Navigate** method already performs basic

normalization internally before it passes the URL to the browser. Use **NormalizeUrl** when you want to enforce additional policy restrictions with the option flags, or notify users of clear validation failures before attempting navigation.

Normalization includes, depending on the flags you choose:

- Scheme defaults and enforcement. If the input omits a scheme, you can default to HTTPS (`URL_FLAG_DEFAULT_HTTPS`) or require http:// or https:// explicitly using `URL_FLAG_REQUIRE_HTTP` or `URL_FLAG_REQUIRE_HTTPS`. If the normalizer cannot accept the scheme under your selected policy, the call fails. The same default behavior is applied with the **Navigate** method.
- Hostname policy. You may reject localhost, numeric IPv4, or bracketed IPv6 literals, or enforce ASCII hostnames (Unicode hostnames are converted to Punycode and lower-cased). The host parsing also ensures the input isn't a Windows path accidentally pasted in place of a URL.
- Path and query normalization. Paths may be percent-encoded as needed, duplicate slashes removed, and path traversal segments are collapsed; queries and fragments may be percent-encoded while preserving common query delimiters.
- Local file handling. Absolute local and UNC paths or existing file:// URLs can be converted to a normalized file:// form unless you explicitly reject them with `URL_FLAG_REJECT_LOCAL_FILES` (and optionally the `URL_FLAG_REJECT_UNC_PATHS` option). If you intend to only permit web content, enable these reject flags. At navigation time the library also tracks whether the target is a local file.
- If your project uses the multi-byte character set instead of Unicode, and `URL_FLAG_VALIDATE_UTF8` is specified, the method will fail if the URL contains any invalid UTF-8 encoding. Without this flag, if the method detects invalid UTF-8 characters, it will attempt to validate the URL using the active code page. It is recommended that applications use UTF-8 or UTF-16 Unicode whenever possible.

This method validates and normalizes URLs but does not expand redirects, verify server reachability, or fetch headers. Use it to block undesired inputs before navigation (such as disallowing bare IPs or local files), or to present a user-friendly, fully qualified URL.

Example

```
// Match the safe normalization options and forbid local files
DWORD dwNormFlags =
    URL_FLAG_DEFAULT_HTTPS |
    URL_FLAG_ASCII_HOSTNAME |
    URL_FLAG_ESCAPE_PATH |
    URL_FLAG_ESCAPE_QUERY |
    URL_FLAG_REJECT_LOCAL_FILES |
    URL_FLAG_REJECT_UNC_PATHS;

// If normalization or navigation fails, save the last error code
DWORD dwError = NO_ERROR;

// Check how long the normalized URL will be
INT nLength = pWebView->NormalizeUrl(pszUserInput, dwNormFlags, NULL, 0);
if (nLength <= 0)
    dwError = pWebView->GetLastError();

if (dwError == NO_ERROR)
```

```
{
    // Allocate buffer and normalize again
    LPTSTR pszNormalizedUrl = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));

    if (!pszNormalizedUrl)
         return; // Failed; virtual memory exhausted

    nLength = pWebView->NormalizeUrl(pszUserInput, dwNormFlags, pszNormalizedUrl,
nLength + 1);
    if (nLength <= 0)
    {
        dwError = pWebView->GetLastError();
    }
    else
    {
        // Navigate using the normalized URL
        if (!pWebView->Navigate(pszNormalizedUrl))
            dwError = pWebView->GetLastError();
    }

    free(pszNormalizedUrl);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetCurrentUrl, Navigate, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::PostWebMessage Method

```
BOOL PostWebMessage (
   LPCTSTR lpszMessage
);
```

The **PostWebMessage** function posts a string or JSON message to the current page.

Parameters

lpszMessage

A null-terminated string which contains the message to be sent to the current page. This parameter cannot be NULL or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **PostWebMessage** method sends a message from the host application to the currently loaded page using the WebView2 messaging channel. The call is asynchronous: if it is invoked from a thread other than the thread that created the browser instance, the method marshals the request to the UI thread and returns immediately, with delivery performed later on that thread. Internally, this posts a private window message to the host window; when handled, it forwards the text using the `ICoreWebView2::PostWebMessageAsString` method.

For a page to receive messages from the host, web messaging must be enabled. When the `WEBVIEW_OPTION_ENABLE_MESSAGES` option is set, the wrapper enables web messages in the control settings (exposing `window.chrome.webview` to the script) and wires the corresponding message events. If this option is not enabled, the page will not see the messaging object and cannot receive messages.

On the JavaScript side, the page subscribes to messages using the standard listener pattern. Messages posted with this method are delivered as strings to the page's handler:

```
window.chrome.webview.addEventListener('message', (event) => {
 // event.data is the string posted by WebViewPostWebMessage
 const text = event.data;
 // If you know the string is JSON, you can parse it explicitly
 // const obj = JSON.parse(text);
});
```

Because delivery is asynchronous, there is no built-in acknowledgement and ordering is not guaranteed across navigations. If the page navigates or reloads after you call **PostWebMessage**, a listener registered on the previous document will not receive later messages.

Messages are Unicode strings and you should avoid sending very large payloads. For large data transfers, consider using application-level protocols rather than embedding large content in a single message. This method is primarily intended to enable your host application to notify the web page the user has taken some action which should be reflected in the browser. The web page can send messages back to your application using `window.chrome.webview.postMessage(value)` and you can receive those messages using the **GetWebMessage** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AsyncExecute, Execute, GetWebMessage, GetWebResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::PrintDocument Method

```
BOOL PrintDocument(
   DWORD dwOptions
);
```

The **PrintDocument** method displays the browser print dialog for the current document.

Parameters

dwOptions

An unsigned integer which specifies how the print dialog should be displayed.

Value	Description
WEBVIEW_PRINT_DIALOG_DEFAULT	Use Edge's print preview dialog, equivalent to the **Ctrl+P** experience. This dialog provides preview capabilities and additional print options.
WEBVIEW_PRINT_DIALOG_SYSTEM	Use the native system print dialog. This dialog allows the user to select a printer and set standard options, but does not provide a preview of the document.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **PrintDocument** method displays the print dialog for the currently loaded document. The dialog is shown asynchronously and the method will return once the print user interface has been invoked. This method does not perform the printing operation itself; it only initiates the print workflow and allows the user to select a printer, adjust preferences, and confirm printing.

This method can only be called after navigation has completed and the current document is fully loaded. If navigation is still in progress when the method is called, it will fail with an error indicating that the operation is in progress. This method only supports interactive printing through a user interface. If you need to programmatically export the document without user interaction, use the **SaveDocument** method instead.

It is recommended that most applications use the default dialog. The `WEBVIEW_PRINT_DIALOG_SYSTEM` option shows a simplified print dialog provided by browser. This is not the same dialog exposed by the Windows Common Controls API through the **PrintDlg** or **PrintDlgEx** functions, and it may differ in appearance and available options depending on the platform and version of the WebView2 runtime.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetDocumentText, GetFilteredText, Navigate, SaveDocument

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Redraw Method

```
BOOL Redraw();
```

The **Redraw** method forces the host window and browser control to redraw.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Redraw** method reloads the current document and forces both the host window and the embedded WebView2 child window to repaint themselves. This differs from **Reload**, which simply reloads the page (similar to pressing **F5**) but does not explicitly invalidate and redraw the hosting window.

Internally, this method clears the browser cache and then forces a redraw by updating the control bounds and notifying the browser that the parent window has changed. If the host window itself is visible, it is also invalidated and updated to ensure that the client area is repainted.

Because this method performs multiple operations that can be relatively expensive, applications should generally prefer **Reload** unless a complete repaint of both the control and the host window is required. For example, it can be used if rendering artifacts are visible after a resize or when switching between different DPI contexts.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCache, Reload, Reset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Reload Method

```
BOOL Reload(
   BOOL bClearCache
);
```

The **Reload** method reloads the current document and optionally clears the cache.

Parameters

bClearCache

An integer which indicates if the document cache for the session should be cleared. A non-zero value specifies the cache should be cleared; zero specifies the cache should not be cleared.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reload** method refreshes the currently loaded page, similar to pressing **F5** in a web browser. If the *bClearCache* parameter is non-zero, the document cache is also cleared before reloading, ensuring that all resources are fetched again from the server instead of being served from the local cache.

This method is generally preferred over **Redraw** when an application only needs to reload the content without forcing the host window and browser control to repaint. It provides a lightweight way to refresh the page and optionally reset cached resources, without the overhead of a full redraw of the hosting window.

An application would typically use this method in response to a user action or when it needs to ensure the latest version of a document is displayed after cached data may have become stale.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCache, Redraw

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Reset Method

```
BOOL Reset();
```

The **Reset** method resets the embedded runtime control.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reset** method clears the state of the current browser session and restores it to its default configuration. This includes discarding cached content, navigation history, and any data associated with the last document. The control is then navigated to an empty `about:blank` page and the zoom level is reset to its default value.

Internally, this method releases cached strings such as the most recent message, script result, and document text, clears the current HTTP status and security information, removes cached data, and clears the session history. A navigation to **about:blank** is performed, and the host window is invalidated to ensure that the client area is repainted. The call blocks until the navigation has completed or the operation times out.

Because this method performs a complete reset of the WebView2 control, it should be used in situations where the browser needs to be returned to a known, clean state (for example, when reinitializing a session or preparing the control for a new task). Applications that only need to refresh the current document should use **Reload** or **Redraw** instead, as they are lighter-weight operations.

This method cannot be called while a navigation is already in progress. Attempting to do so will fail with an error code indicating that the operation is in progress. As with other navigation-related functions, it must be called on the thread that owns the browser window, or the request will be marshaled back to the UI thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

ClearCache, Redraw, Reload

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Resize Method

```
BOOL Resize();
```

The **Resize** method resizes the browser child window to the size of its host window.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resize** method updates the position and size of the embedded browser window to the size of client area of the host window. It calls the ***put_Bounds*** method on the underlying WebView2 controller to resize the child window that renders the web content. The coordinates and dimensions are determined by calling the Win32 function **GetClientRect** with the host window.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Redraw, SetBounds

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::RuntimeAvailable Method

```
static BOOL RuntimeAvailable();
```

The **RuntimeAvailable** method determines whether the required runtime is installed and available.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **RuntimeAvailable** method checks whether the Microsoft Edge WebView2 runtime is installed and meets the minimum required version for this library. If the runtime is available, the method returns `TRUE`; otherwise, it returns `FALSE` and sets the last error code to indicate whether the runtime is missing or is an unsupported version. If the runtime is not installed, the last error code will be set to `ST_ERROR_WEBVIEW2_NOT_INSTALLED`. If the installed runtime is older than the minimum supported version (currently `102.0.1245.22`), the last error code will be set to `ST_ERROR_INVALID_WEBVIEW2_VERSION`.

This is a static method which can be called before creating an instance of the **CWebView** class. It allows an application to check for the availability of the WebView2 runtime at startup, and optionally provide a more helpful error message to the user before attempting to create or attach a browser control. Developers may wish to call this method during application startup or installation checks. For example, if the runtime is not present or is too old, the application can inform the user that they must install or update the Microsoft Edge WebView2 runtime in order to use features which require embedded web content.

If the method succeeds, it only guarantees that the required runtime is installed and is a sufficient version. It does not initialize or configure the runtime; that process occurs inside the **CWebView** class constructor.

Example

```
CWebView *pWebView = NULL;

INT APIENTRY wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR
lpCmdLine, INT nCmdShow)
{
    if (!CWebView::RuntimeAvailable())
    {
        MessageBox(NULL,
                   _T("The Microsoft Edge WebView2 runtime is not available."),
                   _T("MyApp"), MB_ICONERROR | MB_OK);

        return 0; // Exit application
    }

    // Create a new instance of the CWebView class
    pWebView = new CWebView();

    if (!pWebView->IsInitialized())
    {
```

```
        MessageBoxW(NULL,
                    L"Unable to initialize the SocketTools WebView library",
                    L"MyApp", MB_ICONERROR | MB_OK);
        return 0;
    }

    // Create your main window, attach the browser, etc.

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

CWebView, ~CWebView, GetBrowserVersion

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SaveDocument Method

```
BOOL SaveDocument(
   UINT uFileType
   LPCTSTR lpszFileName
);
```

The **SaveDocument** method saves the current document to a local file.

Parameters

uFileType

An unsigned integer which specifies the format the document should be saved as.

Value	Description
WEBVIEW_FILE_TYPE_DEFAULT	Save using a format inferred from the file extension. If the file extension is not recognized, the contents will be saved as plain text. For example, if you specify a file name with the `.html` extension, it will be saved in HTML format. If the file name has a `.pdf` extension, it will be saved as a PDF.
WEBVIEW_FILE_TYPE_TEXT	Saves the current document as plain text. The HTML tags are removed and entities are converted to their UTF-8 equivalent. Extraneous whitespace within the text will be removed, but the overall structure of ordered and unordered lists will be preserved. Choose this format if you wish to save only the text content of a document.
WEBVIEW_FILE_TYPE_HTML	Saves the current document as a UTF-8 encoded HTML file. This includes the complete source for the page, but does not include any of the additional assets such as images. Choose this format if you wish to save the document as an editable HTML file.
WEBVIEW_FILE_TYPE_JSON	Saves the current document as a JSON formatted string. The string will be quoted and reserved characters will be escaped. Choose this format if you wish to preserve the HTML document as a JSON string for logging or embedding in a JSON object.
WEBVIEW_FILE_TYPE_MHTML	Saves the current document as a single-file web archive (MHTML) that inlines the HTML content and all of the resources it references. This format calls the DevTools `Page.captureSnapshot` method in the browser and writes the returned data to disk. MHTML is useful for offline archiving since it combines images, styles and scripts together in a single file.
WEBVIEW_FILE_TYPE_PDF	Saves the current document as a PDF file which produces a paginated document suitable for printing and distribution. This is functionally similar to using the browser to print to a PDF, but does not display a dialog to the user.

WEBVIEW_FILE_TYPE_JPEG	Save the current document view as a JPEG image. JPEG is a widely used image format optimized for photographs and complex images with smooth gradients. It uses lossy compression to significantly reduce file size while maintaining acceptable visual quality.
WEBVIEW_FILE_TYPE_PNG	Save the current document view as a PNG image. PNG is a lossless image format designed for crisp detail and high fidelity, especially in graphics with sharp edges, text, or flat colors. Although PNG files are typically larger than JPEGs, they preserve exact image data without quality loss.
WEBVIEW_FILE_TYPE_WEBP	Save the current document view as a WebP image. WebP is a modern image format developed by Google that supports both lossy and lossless compression, as well as transparency and animation. It delivers smaller file sizes compared to JPEG and PNG while maintaining comparable or better visual quality.

lpszFileName

A null-terminated string which specifies the name of the local file. If the path includes environment variables enclosed in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL and must specify a valid file path. The file name cannot specify a device, folder name or system file. If the file is marked as read-only, the method will fail.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SaveDocument** method saves the current page to a local file in the format specified by the *uFileType* parameter. The operation is synchronous, processing messages while the browser completes the action, and only returns once the document has been created or the operation has failed. A temporary file is created during the save process and then moved into place, replacing any existing file. This ensures that an existing file will not be changed unless the browser indicates the save operation was successful. Saving is not permitted while a navigation is in progress, and attempts to do so will fail with an error.

The file name specified must be valid, and environment variables in the path are expanded before use. When **WEBVIEW_FILE_TYPE_DEFAULT** is specified, the format is inferred from the file extension. For example, **.html** or **.htm** will save the document as HTML, **.pdf** will create a PDF file and **.png** will save the page as a PNG image. If no recognized extension is present, the page will be saved as plain text by default.

When saving the current document as plain text, the HTML markup will be removed and it will be saved as a UTF-8 encoded text file. When saving as HTML, the method saves the complete HTML source for the page. The JSON format also saves the HTML markup, but encodes it as a quoted JSON string with reserved characters escaped. Note that all text-related formats are saved using UTF-8 multi-byte encoding without a byte order mark (BOM). The text will not be automatically converted to the current user's locale. Most modern text editors automatically recognize UTF-8 encoding, but some legacy applications may assume the text is localized to the current code page.

The MHTML option creates a single-file archive which contains the page content and all of its resources such as images, style sheets, and scripts. This format is often used for offline archiving or sharing since everything is contained in one file. Saving as PDF renders the page into a paginated print-style format suitable for distribution. These formats all operate on the complete document, including content that extends beyond the visible area.

By contrast, saving as an image (JPEG, PNG, or WebP) captures only the visible viewport of the page at the time the method is called. Content that requires scrolling will not be included in the saved image. PNG is lossless and preserves transparency, while JPEG and WebP use compression to produce smaller files. WEBP generally offers better quality at a reduced size compared to JPEG, but support for the format may vary between applications. If you want a pixel-perfect representation of the current page as it appears on the screen, particularly if it contains a lot of text, it is recommended that you use the PNG format.

Because this method blocks the current thread until the save completes, it may take a noticeable amount of time to return if the current document is very large. If the method is called from a worker thread, the request is marshaled to the UI thread that manages the browser control. For applications with a graphical interface, it is recommended that you display the standard hourglass (or other busy indicator) while the save is in progress to provide a visual cue to the user.

If you want a copy of the HTML or plain text contents of the current page without saving it to a file, use the **GetDocumentText** or **GetFilteredText** methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDocumentText, GetFilteredText, Navigate, PrintDocument, SetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetBackgroundColor Method

```
BOOL SetBackgroundColor(
   COLORREF rgbBackground
);
```

The **SetBackgroundColor** method changes the background color used behind the page content.

Parameters

rgbBackground

An integer value which specifies the background color.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The background color specifies the fill used by the browser window when there is no content to display or the document itself does not define a background. This value only applies to the WebView host surface and does not affect page transparency. The alpha channel is ignored, and the color is treated as fully opaque.

It is important to note that this setting does not override the background defined by the currently loaded page. If the page specifies its own background color or image, that content will take precedence and may obscure the configured color. For example, setting the background to white will not force web pages to appear white unless the page itself has transparent regions.

Applications can use this method to provide a consistent appearance for empty or loading views, but should not rely on the background color as a substitute for page styling. When a browser instance is first created, the default background color is white.

A **COLORREF** is a 32-bit value used by the Windows API to represent an RGB color. The low-order byte specifies the red component, the next byte the green component, and the third byte the blue component, in the format **0x00BBGGRR**. The high-order byte is unused and should be zero. Note that **COLORREF** is specific to the Windows API and is not the same as the OLE color system (**OLE_COLOR**). OLE uses a slightly different encoding that allows system color values to be specified, which are resolved at runtime. Developers should be careful not to confuse the two.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetBackgroundColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetBounds Method

```
BOOL SetBounds(
   LPRECT lpRect
);

BOOL SetBounds(
   INT xPos,
   INT yPos,
   INT nWidth,
   INT nHeight
);
```

The **SetBounds** method sets the position and size of the browser child window.

Parameters

lpRect

A pointer to a `RECT` structure which specifies the position and dimensions of the browser window. If this version of the method is called, this parameter cannot be NULL.

xPos

The x-position of the child window. This is a client area coordinate of the host window, not a screen position.

yPos

The y-position of the child window. This is a client area coordinate of the host window, not a screen position.

nWidth

The width of the child window. This is expressed in device units for the host window's client area.

nHeight

The height of the child window. This is expressed in device units for the host window's client area.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetBounds** method updates the position and size of the embedded browser window relative to the client area of the host window. It calls the ***put_Bounds*** method on the underlying WebView2 controller to resize the child window that renders the web content. The specified coordinates and dimensions must fall within the client area of the host window associated with the browser instance. Attempting to specify bounds outside of the host window will result in clipping or the control being partially hidden.

The *nWidth* and *nHeight* parameters are device units, the same pixel units that the Win32 API uses with functions like **MoveWindow** and **SetWindowPos**. If the host window was created as a per-monitor DPI-aware window, the values your application gets with a `WM_SIZE` message are already scaled for the current DPI and you should just pass those values directly to this method. If your application is not DPI-aware, Windows will attempt to scale the values for you through DPI virtualization. It is important to note that these values are not logical units like Twips and they are not independent of DPI scaling.

Typically this method is called from within a **WM_SIZE** or **WM_MOVE** message handler so that the browser control resizes and repositions itself along with the parent window. You can also call this function directly to dock the browser into a specific portion of the host window's client area, such as aligning it within a child frame or panel.

In most applications it is sufficient to call this method in response to a **WM_SIZE** message so the browser control resizes along with the client area of the host window. However, in a per-monitor DPI-aware application it may also be useful to call this method when handling **WM_MOVE**. This is because moving a top-level window between monitors can trigger a DPI change even if the window size itself does not change. Reapplying the bounds in response to **WM_MOVE** ensures the control is correctly adjusted to the new scaling factor. For applications that are not DPI-aware, handling **WM_MOVE** is generally unnecessary since the client area origin remains at (0,0) and Windows applies DPI virtualization automatically.

Calling **SetBounds** does not affect the visibility of the browser window. To hide or show the control, use the **Hide** or **Show** methods, and to query its visibility, use **IsVisible**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetFocus, Hide, IsVisible, Redraw, SetFocus, Show

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetDocumentText Method

```
BOOL SetDocumentText(
   LPCTSTR lpszText
);
```

The **SetDocumentText** method replaces the current document with the specified HTML or plain text.

Parameters

lpszText

A null-terminated string which specifies the HTML or plain text that should replace the current document. If this parameter is NULL the current document will be cleared.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetDocumentText** method replaces the currently displayed content with the specified HTML or plain text. If the text does not contain any markup, it will be wrapped in HTML as preformatted text and displayed using the default monospaced font. If the *lpszText* parameter is `NULL` or an empty string, the browser will display a blank page.

The string passed to this method is normally rendered directly by the browser. If the string is very large (over 256 KiB), the library will first write the text to a local temporary file and then instruct the browser to navigate to that file. This is done to work around limitations in the Edge WebView2 control when handling long inline strings. The temporary file will be automatically deleted when it is no longer needed.

If your project is configured to use a multi-byte character set (ANSI) instead of Unicode, this method will always interpret the *lpszText* string as UTF-8 encoded text, regardless of the system's current locale or code page. Legacy ANSI code pages cannot reliably represent the full range of characters, and using them in place of Unicode or UTF-8 can lead to data loss or corrupted text.

Important: It is possible to embed markup that references images, scripts, or other resources located on the Internet. While this is supported, you should be aware of potential risks when combining local content with external resources. This can create mixed-content scenarios and introduce cross-origin or security issues. Applications that render untrusted content should avoid linking to remote resources and ensure that the HTML provided is properly validated.

Example

In this example, the method replaces the current page with a basic HTML document. The `<img>` tag references an external image from a website. This is supported, but as noted above, applications should carefully consider the security implications of mixing local content with external resources.

```
// Replace the current document with a simple HTML page
LPCTSTR pszHtml =
    _T("<html><head><title>Hello</title></head>")
    _T("<body><h1>Hello, world!</h1>")
    _T("<p>This content was created dynamically.</p>")
    _T("<img src=\"https://example.com/logo.png\" alt=\"Logo\"/>")
    _T("</body></html>");

if (!pWebView->SetDocumentText(pszHtml))
```

```
{
    DWORD dwError = pWebView->GetLastError();
    // Handle the error
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetDocumentText, GetDocumentTitle, GetFilteredText, Navigate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetDownloadPath Method

```
BOOL SetDownloadPath(
   DWORD dwFileId,
   LPCTSTR lpszFilePath
);
```

The **SetDownloadPath** method changes the location of the local file for the specified download.

Parameters

dwFileId

An unsigned integer value which uniquely identifies the download. This value cannot be zero.

lpszFilePath

A null-terminated string which specifies the new path for the downloaded file. This value may be a `file://` URL or it may specify a local file path. If the path contains environment variables surrounded with percent (%) symbols, the variables will be expanded. This parameter cannot be NULL or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The specified path may be provided as either a file:// URL or a local file system path. If a file:// URL is given, it will be converted to a local path before use. Environment variables enclosed in percent signs (such as `%TEMP%`) are expanded automatically. If the path refers to an existing directory rather than a file name, the original file's base name is preserved and the file will be created in that directory. This allows applications to easily redirect a download into a specific folder without changing the file name.

The method always normalizes the provided path. If a relative path is specified, it is resolved against the current working directory of the process. Because this can lead to unpredictable results in multithreaded applications, it is strongly recommended that applications always provide absolute paths. By default, downloaded files are typically stored in the current user's Downloads folder.

Several restrictions are enforced on the path to help ensure safety and compatibility. Device names such as `CON`, `PRN`, or `NUL` are not permitted. Downloads cannot be redirected into system folders such as the Windows folder or Program File folder. The path cannot contain reserved characters, and files that are marked as read-only will not be overwritten. If any of these conditions are detected, the method will fail with an error.

This method can only be called within a handler for the `WEBVIEW_EVENT_DOWNLOAD_STARTING` event. Once the download has begun, the local file path cannot be changed. If you wish to move the file after it has been downloaded, you can use the **GetDownloadPath** method to obtain its current location and then use a Win32 function like **MoveFileEx** to move the file to a new folder.

Note that you should avoid showing modal dialogs such as the common "Save File" dialog within event handlers. Because the browser events are raised within the context of the WebView2 UI thread, displaying a dialog asking the user for a new download location can prevent the control from processing its internal message loop, potentially leading to stalled downloads.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CancelDownload, EnumDownloads, GetDownloadInfo, GetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetFocus Method

```
BOOL SetFocus();
```

The **SetFocus** method changes focus to the browser control.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetFocus** method gives input focus to the embedded browser control. Once focus has been set, keyboard input such as typing or navigation keys will be directed to the content displayed by the control rather than to other controls in the host window. This is equivalent to the user clicking inside the browser area.

Setting focus does not make the host window itself the active foreground window. If the host window is not active, input will still not be directed to the browser until the window is activated by the user or programmatically. Applications that need to determine whether the browser currently has focus can call **GetFocus**.

Example

```
BOOL bHasFocus = FALSE;

if (pWebView->GetFocus(&bHasFocus))
{
    if (!bHasFocus)
    {
        // Give the browser control focus if it does not have it
        pWebView->SetFocus();
    }
}
else
{
    // Handle error
    DWORD dwError = pWebView->GetLastError();
    _tprintf(_T("GetFocus failed, error=0x%lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetFocus, GetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetHeader Method

```
BOOL SetHeader (
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue
);
```

The **SetHeader** function sets a request header value to be included with subsequent navigations.

Parameters

lpszHeaderName

A null-terminated string which specifies the name of the request header value to set or change.

lpszHeaderValue

A null-terminated string which specifies the value of the header. If this value is NULL or an empty string, the request header will be removed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetHeader** method adds or updates a custom request header that will be included with subsequent navigations. Request header names are not case-sensitive, and any leading or trailing whitespace is ignored. If the specified header value is NULL or an empty string, the header will be removed from the current list of custom headers. These changes only affect future requests; they do not modify headers already sent for the current navigation.

Certain standard headers are controlled by the underlying WebView2 control and may be ignored if set programmatically. Examples include `Host`, `Referer`, `Origin`, `User-Agent`, and most authentication headers. Attempting to override these may have no effect. For headers that can be set, the specified values will be merged with the default request headers used by WebView2 when making requests. If you wish to change the `User-Agent` header value, use the **SetUserAgent** method.

When defining custom request headers, it is recommended to use names prefixed with X- (for example, `X-Application-ID` or `X-Request-Token`). This convention helps distinguish application-specific headers from those defined by the HTTP standard, and reduces the likelihood of conflicts with headers that are managed internally by WebView2 or intermediaries such as proxies and gateways. While most modern systems will accept arbitrary header names, following this convention improves compatibility and avoids unexpected behavior.

If you need to clear all custom request headers that have been set, call **ClearHeaders** with the `WEBVIEW_HEADERS_REQUEST` option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ClearHeaders, GetFirstHeader, GetHeader, GetNextHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetIcon Method

```
BOOL SetIcon(
   HINSTANCE hResInst,
   LPCTSTR lpszIconName,
   UINT uResId,
   DWORD dwFlags
);

BOOL SetIcon(
   LPCTSTR lpszIconName,
   DWORD dwFlags
);
```

The **SetIcon** method sets custom large and small icons for the host window.

Parameters

hResInst

Handle to the module instance that contains the icon resource. If this parameter is NULL, the *lpszIconName* parameter should specify the path to an icon file.

lpszIconName

A null-terminated string that specifies the name of the icon resource or a path to an icon file. If this parameter is not NULL, it takes precedence over *uResId*.

uResId

An unsigned integer which specifies the numeric identifier of the icon resource within the module. This is used when *lpszIconName* is NULL.

dwFlags

An unsigned integer which specifies which icons to update. This must be one or more of the values shown in the following table.

Value	Description
WEBVIEW_ICON_BIG	Update the large window icon.
WEBVIEW_ICON_SMALL	Update the small window icon.
WEBVIEW_ICON_BOTH	Update both large and small icons.
WEBVIEW_ICON_DPI_AWARE	Choose the system-recommended small icon size for the current DPI when possible.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetIcon** method updates the large and/or small icons associated with the host window created using the **Create** or **Create** method. This is equivalent to sending the `WM_SETICON` message to the host window, but automatically handles both the large and small icons and supports DPI-aware sizing when requested.

If the *hRetInst* parameter is not NULL, either a string resource name (*lpszIconName*) or a numeric

resource identifier (*uResId*) can be specified to identify the icon. If both are provided, the string resource name takes precedence. The resource must exist in the module instance specified by *hResInst*. If neither *lpszIconName* nor *uResId* resolves to a valid icon, the method fails.

if the *hResInst* parameter is NULL, the *lpszIconName* parameter must specify the name of a file which contains the icon and the *uResId* parameter is ignored. It is recommended that you provide an absolute path to the icon file. If the path includes environment variables surrounded with percent (%) symbols, those variables will be automatically expanded. The file must use the standard Windows icon format.

The *dwFlags* parameter determines which icons are updated. You can set the large icon (used in the Alt-Tab interface and title bar), the small icon (shown in the window caption and taskbar), or both. If `WEBVIEW_ICON_DPI_AWARE` is specified, the system will choose an appropriate small icon size based on the current display scaling.

Windows icons are stored in the standard `.ico` file format, which can contain one or more images at different resolutions and color depths. When you compile icons into your application's resources and reference them by name or identifier, Windows will automatically select the most appropriate image for the current display settings. This allows a single icon resource to provide 16x16, 32x32, 48x48, or higher-resolution images within the same file.

The operating system chooses which image to use based on context and DPI scaling. For example, a 16x16 version may be displayed in a window caption or taskbar, while a 32x32 or 48x48 version may be shown in the Alt-Tab interface or Windows Explorer. If `WEBVIEW_ICON_DPI_AWARE` is specified, the system can pick a small-icon size that best matches the current DPI rather than always defaulting to 16x16.

Calling this method does not destroy the icon resource itself; ownership remains with the module that loaded it. If you dynamically create icons at runtime using **CreateIcon** or **LoadImage**, you are responsible for releasing them with **DestroyIcon** if they are no longer needed.

This method can only be used with windows created by this library, it cannot be used with windows you attach the browser to using the **Attach** method.

Example

```
// Define the path to the application's icon file
LPCTSTR pszIconFile = _T("%LocalAppData%\\MyApp\\MyApp.ico");

if (pWebView->Create(_T("about:blank"))
{
    // Set both large and small icons for the WebView host window
    pWebView->SetIcon(pszIconFile, WEBVIEW_ICON_BOTH | WEBVIEW_ICON_DPI_AWARE);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Create

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE or NULL. Those functions which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetErrorString, GetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetOption Method

```
BOOL SetOption(
   DWORD dwOption,
   BOOL bEnable
);
```

The **SetOption** method enables or disables a specific option.

Parameters

dwOption

An unsigned integer which specifies the option to be changed. Refer to the Options documentation for the list of available options. Only a single option may be specified; you cannot combine multiple option bitflags when calling this method.

bEnable

An integer value which specifies if the option should be enabled. If this value is non-zero, the option is enabled. If the value is zero, the option is disabled.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetOption** method enables or disables a single option for the browser instance attached to the specified host window. Each option corresponds to a specific feature or behavior such as scripting, zoom controls, developer tools, or privacy settings. Only one option bitflag may be specified in the *dwOption* parameter when calling this method. It is not valid to combine multiple option flags with a bitwise OR. If you need to change the state of multiple options, call this method once for each option you want to modify.

Some options can be changed dynamically at runtime, while others only take effect during creation of the browser control. For example, options related to scripting, dialogs, or developer tools can typically be toggled while the control is active. In contrast, options that affect the user data folder, profile isolation, or privacy mode must be set at creation time using **Attach** or **Create**. Attempting to change those options will fail once the browser control has been created.

If the method succeeds, the option state will be updated immediately. The application can verify the new state by calling **GetOption** or by retrieving all current options with **GetAllOptions**. If the method fails, no changes are made and extended error information can be obtained with **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, GetAllOptions, GetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetTimeout Method

```
INT SetTimeout(
   INT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the browser will wait for a response from the server.

Parameters

nTimeout

The number of seconds to wait for the navigation to complete.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is WEBVIEW_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetTimeout** method specifies the maximum number of seconds that the library should wait for a navigation request to complete. This timeout begins when navigation starts and ends when the navigation completes or fails.

If the timeout period is set to zero, the library does not enforce its own timer and instead relies on the default timeouts implemented by the WebView2 engine. These built-in timeouts automatically handle conditions such as slow DNS resolution, stalled connections, or unreachable servers, and generally behave in the same way as the Microsoft Edge browser.

If a non-zero timeout period is specified, the control will cancel the navigation if the internal timer exceeds that value and the page has not finished loading. When this occurs, the navigation is stopped and a timeout error is reported through the standard navigation event notifications. This allows applications to fail fast in cases where a site is not responding, rather than waiting for the browser's adaptive timeouts to expire.

Note that specifying a very large timeout value will not force the browser to wait indefinitely. The WebView2 engine may still apply its own internal maximums and terminate the navigation earlier if it decides the request cannot succeed. Applications should select timeout values that balance responsiveness and reliability.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, GetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetUserAgent Method

```
BOOL SetUserAgent(
   LPCTSTR lpszUserAgent
);
```

The **SetUserAgent** method changes the default User-Agent string for this instance.

Parameters

lpszUserAgent

A null-terminated string which specifies the new User-Agent header value for subsequent requests. If this parameter is NULL or an empty string, the User-Agent header will revert back to the default value used by the Edge WebView2 control.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetUserAgent** method sets the string that will be sent in the User-Agent header for all subsequent HTTP requests made by this browser instance. This method should always be used to modify the User-Agent header value; you should not set the header directly using the **SetHeader** function.

Applications may want to override the default User-Agent in several scenarios:

- Requesting a mobile or tablet version of a site by providing a User-Agent string that identifies as a mobile browser.
- Emulating a different browser for testing purposes.
- Including custom product tokens identifying the host application to the server.

The general format of a User-Agent string follows the conventions described in RFC 9110:

```
User-Agent: product / version (comment) [ product / version ... ]
```

For example, a typical User-Agent string might look like:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/140.0.0.0 Safari/537.36 Edg/140.0.0.0
```

This value is global to the instance. Once changed, it will remain in effect until explicitly reset or until the browser window is destroyed. When setting a custom User-Agent value, developers should ensure it is a syntactically valid string. Some websites may reject requests or provide degraded functionality if the value is malformed or omits key tokens.

If you want to change the User-Agent string to identify your application, it is recommended that you append your application name and version to the default value.

Example

```
// First, get the length of the current User-Agent string
INT nLength = pWebView->GetUserAgent(NULL, 0);
if (nLength <= 0)
{
    _tprintf(_T("Unable to get User-Agent string (error 0x%lx)\n"), pWebView-
```

```
>GetLastError());
    return;
}

// Allocate a buffer large enough to append the application name and version;
// note the leading space in the name to separate it from other tokens
const TCHAR szAppId[] = _T(" MyApp/1.0");

INT nTotalLength = nLength + (INT)_tcslen(szAppId) + 1;
LPTSTR lpszUserAgent = (LPTSTR)malloc(nTotalLength * sizeof(TCHAR));

if (!lpszUserAgent)
    return;

// Retrieve the current User-Agent value from the browser
if (pWebView->GetUserAgent(lpszUserAgent, nTotalLength) <= 0)
{
    _tprintf(_T("Failed to copy User-Agent string (error 0x%lx)\n"), pWebView-
>GetLastError());
    free(lpszUserAgent);
    return;
}

// Append our application identifier to the User-Agent string
_tcscat_s(lpszUserAgent, nTotalLength, szAppId);

// Update the User-Agent header with the new value
if (!pWebView->SetUserAgent(lpszUserAgent))
{
    _tprintf(_T("Failed to update the User-Agent string (error 0x%lx)\n"),
pWebView->GetLastError());
}
else
{
    _tprintf(_T("User-Agent successfully updated to:\n%s\n"), lpszUserAgent);
}

free(lpszUserAgent);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetHeader, GetUserAgent, SetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::SetZoomLevel Method

```
BOOL SetZoomLevel(
   INT nPercent
);
```

The **SetZoomLevel** method changes the current browser zoom level.

Parameters

nPercent

An integer which specifies the zoom level as a percentage. The minimum zoom level is 25 and the maximum is 400.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetZoomLevel** method changes the zoom level of the current browser window to the specified percentage. A zoom level of **100** represents the default size with no scaling applied. Larger values magnify the content, while smaller values reduce it.

The zoom percentage is constrained by the limits defined for the control. The minimum supported zoom level is **25** (25%) and the maximum is **400** (400%). If a value outside of this range is specified, it will be clamped to the nearest valid zoom level.

Changing the zoom level affects how the page content is displayed but does not alter the underlying document. This is equivalent to adjusting the zoom in a standard web browser. The new zoom level applies to the current browser instance and will remain in effect until it is changed again, or the instance is destroyed.

If the method fails, the zoom level remains unchanged. To query the current zoom level, call **GetZoomLevel**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::Show Method

```
BOOL Show();
```

The **Show** method shows the browser window.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Show** method changes the visibility state of the browser control. When the browser was created as a top-level window using **Create**, this method internally calls the standard **ShowWindow** API to show the entire browser window.

When the browser is embedded in an existing host window using **Attach**, the method does not change the visibility of the parent window. Instead, it sends an internal message to the control to show the child window which contains the WebView2 browser. This enables the host application to enable visibility of the browser control independently of other controls in the same window.

The current visibility state can be queried using the **IsVisible** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

GetFocus, Hide, IsVisible, Redraw, SetBounds, SetFocus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::WaitForEvent Method

```
BOOL WaitForEvent(
   UINT uEventId,
   DWORD dwMilliseconds
);
```

The **WaitForEvent** method waits until a specific event occurs or the timeout period expires.

Parameters

uEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the Events documentation.

dwMilliseconds

The number of milliseconds to wait for the event to occur. If this amount of time elapses, the method returns. If this value is zero, the method will return successfully if the last event handled matches the ***uEventId*** parameter. This can be used to effectively poll the browser to determine if the specified event has occurred. This value may be `INFINITE` in which case the function will not return until the event occurs.

Return Value

If the method succeeds, the return value is non-zero. If the method fails or the timeout period is exceeded, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WaitForEvent** method suspends the calling thread until the specified event is raised or the timeout interval expires. While waiting, this method continues to process the thread's Windows message queue. This ensures that the user interface remains responsive and that event callbacks are able to execute on the correct thread.

Because messages are dispatched while this method blocks, application code must be written with the possibility of reentrancy in mind. That is, another message handler or event callback may run before **WaitForEvent** returns. If your application maintains shared state, you should use appropriate synchronization (critical sections, mutexes, etc.) to ensure it remains consistent.

This method should generally be called from the same thread that created the browser instance. Calling it from a worker thread that does not own a message loop may prevent events from being delivered. If you require a fully non-blocking design, you should use the event callback function specified when creating or attaching the browser window, rather than waiting synchronously.

You should not call **WaitForEvent** from within an event handler. Doing so can result in unexpected behavior, as the method will attempt to wait for additional events and dispatch messages while another event is already being processed. This may lead to subtle reentrancy issues or even deadlocks depending on the event ordering. Event handlers should return promptly, and synchronous waits should be performed only outside of callback contexts.

Note that if ***dwMilliseconds*** is set to `INFINITE`, the method will not return until the event occurs. To avoid unresponsive behavior, it is recommended you use a finite timeout value unless you are certain the event will always be signaled within a reasonable period.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, WaitUntilClosed, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView::WaitUntilClosed Method

```
BOOL WaitUntilClosed(
   DWORD dwMilliseconds
);
```

The **WaitUntilClosed** method waits until the host window is closed or destroyed.

Parameters

dwMilliseconds

The number of milliseconds to wait for the browser window to close. If this amount of time elapses, the method returns. If this value is zero, the method will not enter a wait state and returns immediately. This value may be `INFINITE` in which case the function will not return until the window is closed.

Return Value

If the browser window has been closed, the return value is non-zero. If the timeout period has been exceeded or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WaitUntilClosed** method blocks the calling thread until the browser window created by **Create** has been closed or destroyed, or until the specified timeout interval has elapsed. Internally, this method waits on the thread handle associated with the WebView2 instance using the `WaitForSingleObject` function.

This method can only be used with browser windows that were created by this class using the **Create** method. It is not valid to call this method with an instance that was attached to a window using **Attach**. Attempting to do so will cause the method to fail.

Unlike **WaitForEvent**, this method does not process any Windows messages while it is waiting. As a result, it should not be used by applications that have a message loop, such as GUI programs that create windows with menus, controls, or dialog boxes. Because messages are not dispatched, the application's user interface may become unresponsive if this method is called from the primary UI thread.

This method is intended primarily for use in console applications or background worker threads, where it may be necessary to determine when the browser window has been closed by the user. GUI applications should use the `WEBVIEW_EVENT_WINDOW_DETACHED` event notification instead of calling this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Create, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewEventProc Function

```
VOID CALLBACK WebViewEventProc(
    HWND hWndHost,
    UINT uEventId,
    WPARAM wParam,
    LPARAM lParam,
    DWORD_PTR dwParam
);
```

The **WebViewEventProc** function is an application-defined callback function that processes events generated by the browser.

Parameters

hWndHost

Handle to the client session.

uEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the Events documentation.

wParam

An integer value that depends on the event context. This value is commonly used by an event to indicate success (1) or failure (0). It may also be used to indicate a transition in state. Refer to the events documentation for information about how this parameter may be used by a specific event.

lParam

An integer value that depends on the event context. This value is commonly used by an event to provide a status code or error code, depending whether the operation which raised the event succeeded or failed. Refer to the events documentation for information about how this parameter may be used by a specific event.

dwParam

A user-defined integer value which was provided when the event callback was specified. This is typically used to provide additional context to the event or associate the event with a specific object. If this user-defined parameter was not specified when the browser window was attached or created, this value will be zero.

Return Value

None.

Remarks

The class invokes the callback function whenever an event occurs. Applications are not required to implement event handlers for each event, and may choose to not provide any event handlers by passing NULL as the callback function parameter. A minimal implementation would typically create handlers for the `WEBVIEW_EVENT_NAVIGATION_STARTING` event which notifies the application when the browser is beginning to navigate to a new location, and the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event which indicates the navigation has completed.

When you attach a browser instance to an existing window, the callback function is always invoked on the same UI thread that owns that window, ensuring events are processed in the context of the caller's thread. However, if you create a new browser window using the **Create** method, a different thread manages the top-level window and message loop, and in that case the callback function is invoked on

the thread that owns the browser window rather than the thread that created it. This distinction is important when your application accesses shared data or resources from within the event handler. To avoid race conditions or undefined behavior, ensure that access to shared state is synchronized using mutexes, critical sections, or other appropriate synchronization primitives.

It is important to keep event handlers responsive. Because events are raised on the UI thread, long-running work in an event handler can block the user interface and stall navigation. Where possible, defer heavy operations to background threads. Avoid displaying modal dialogs or message boxes from within event handlers. Also note that some events may fire multiple times during a single navigation. For example, the security state can transition through several levels (unknown, neutral, secure, or warning) before settling on a final value. Applications should be written to handle these changes incrementally.

After the browser has been attached to a window, it is possible to change the event handler using the **EnableEvents** function. This changes the event handler for the browser instance, and from that point forward all event notifications will be routed to the new handler.

When defining an event handler within a C++ class, the event callback must be declared as a static function with the **__stdcall** calling convention. This is required because the library calls the handler just as it would with any Win32 API callback function, without knowledge of C++ instance objects. Non-static member functions include a hidden **this** pointer parameter, which makes their signature incompatible with the expected **WEBVIEWEVENTPROC** type. A common pattern is to pass the object's ***this*** pointer as the ***dwParam*** value, then cast it back inside the static callback function to access the instance. This preserves object-oriented design while still meeting the calling convention requirements of the API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

Attach, Create, DisableEvents, EnableEvents, FreezeEvents, InEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Data Structures

- WEBVIEW_DOWNLOAD_INFO
- WEBVIEW_SECURITY_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEBVIEW_DOWNLOAD_INFO

The **WEBVIEW_DOWNLOAD_INFO** structure contains information about the current state of a file download.

```
typedef struct _WEBVIEW_DOWNLOAD_INFO
{
    DWORD dwSize;
    DWORD dwFileId;
    DWORD dwStatus;
    ULARGE_INTEGER uiBytesTotal;
    ULARGE_INTEGER uiBytesCopied;
    DWORD dwTimeElapsed;
    TCHAR szFileName[260];
    TCHAR szFileUrl[2048];
} WEBVIEW_DOWNLOAD_INFO, *LPWEBVIEW_DOWNLOAD_INFO;
```

Members

dwSize

Specifies the size of the structure in bytes. This member must be initialized by the caller.

dwFileId

A unique identifier assigned to the download. This value can be used with functions such as **GetDownloadInfo** and **CancelDownload** to query or manage the download.

dwStatus

Indicates the current state of the download. This will be one of the following values:

Name	Description
WEBVIEW_DOWNLOAD_STARTED	The file download has started and is in-progress.
WEBVIEW_DOWNLOAD_COMPLETED	The file download has completed successfully.
WEBVIEW_DOWNLOAD_CANCELED	The file download has been canceled.
WEBVIEW_DOWNLOAD_FAILED	The file download failed due to a network error or other problem.

uiBytesTotal

The total size of the file, in bytes, as reported by the server. If the server did not provide the file size, this value may be zero.

uiBytesCopied

The number of bytes that have been downloaded so far. If the download has completed, this value will be equal to ***uiBytesTotal***. If the total size is unknown, this value still reflects the number of bytes received.

dwTimeElapsed

The elapsed time for the download, in milliseconds. For completed downloads this is the total time from start to finish. For active downloads this is the time since the download began. If the download has not started, this value is zero.

szFileName

A null-terminated string which specifies the local file name associated with the download. If the application has not changed the file path using **SetDownloadPath**, this will be the default file name

assigned by the browser. Typically the file will be stored in the current user's Downloads folder.

szFileUrl

A null-terminated string which specifies the URL of the file being downloaded. This is the fully qualified resource location provided to the browser.

Remarks

This structure provides detailed information about the progress and final state of a download managed by the browser. Before calling a function that populates this structure, such as **GetDownloadInfo**, the caller must set the ***dwSize*** member to the size of the structure. This ensures that applications remain compatible if new fields are added in future updates.

The ***uiBytesTotal*** and ***uiBytesCopied*** members allow applications to report download progress or log completed downloads. If the total file size is not known, ***uiBytesTotal*** will be zero and applications should base their progress reporting on the number of bytes copied alone.

The ***dwFileId*** member can be obtained from **WebViewEnumDownloads**, and is required to retrieve download information or to cancel a pending download.

Both ANSI (A) and Unicode (W) versions of this structure are provided. Applications that call **GetDownloadInfoW** will populate the wide-character members of the structure (such as ***szFileName*** and ***szFileUrl***) with UTF-16 encoded text. Applications that call **GetDownloadInfoA** will receive the ANSI version of the structure, where those same members are returned as UTF-8 encoded strings. The default **WEBVIEW_DOWNLOAD_INFO** macro maps to one or the other depending on whether the UNICODE preprocessor symbol is defined at compile time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Unicode: Implemented as Unicode and ANSI versions

See Also

CancelDownload, EnumDownloads, GetDownloadInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEBVIEW_SECURITY_INFO

The **WEBVIEW_SECURITY_INFO** structure contains information about the current state of a secure connection.

```
typedef struct _WEBVIEW_SECURITY_INFO
{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwLevel;
    TCHAR szProtocol[32];
    TCHAR szCipher[64];
    TCHAR szSubject[128];
    TCHAR szIssuer[128];
    FILETIME ftValidFrom;
    FILETIME ftValidTo;
} WEBVIEW_SECURITY_INFO, *LPWEBVIEW_SECURITY_INFO;
```

Members

dwSize

Specifies the size of the structure in bytes. This member must be initialized by the caller.

dwFlags

A bitmask that provides additional detail about the connection and certificate state. One or more values may be set to indicate conditions such as unencrypted content, use of weak algorithms, or general certificate errors. It may be one or more of the security flags listed below.

Name	Description
WEBVIEW_SECURITY_FLAG_NONE	No additional security indicators.
WEBVIEW_SECURITY_FLAG_UNENCRYPTED	Content is not encrypted.
WEBVIEW_SECURITY_FLAG_INSECURE	Site is insecure using outdated protocols or weak ciphers.
WEBVIEW_SECURITY_FLAG_CERT_WEAK	The certificate uses weak signatures.
WEBVIEW_SECURITY_FLAG_CERT_ERROR	The certificate is invalid. This is a general error condition.
WEBVIEW_SECURITY_FLAG_CERT_EXPIRED	The certificate expired but is otherwise valid.
WEBVIEW_SECURITY_FLAG_UNSAFE	The site is considered potentially unsafe (it failed reputation checks).

dwLevel

Overall security level for the current page. This is a summarized assessment intended for UI decisions, such as showing a lock icon or warning. It maps to one of the security level values listed below.

Name	Description
WEBVIEW_SECURITY_LEVEL_UNKNOWN	The security state could not be determined.
WEBVIEW_SECURITY_LEVEL_INSECURE	The page was retrieved using plain HTTP or the site has an invalid certificate.

WEBVIEW_SECURITY_LEVEL_WARNING	The page contains mixed content or only weak encryption is available.
WEBVIEW_SECURITY_LEVEL_NEUTRAL	Local content or the browser is in a transitional/loading state.
WEBVIEW_SECURITY_LEVEL_SECURE	The page was retrieved using HTTPS and the site has a valid certificate.

szProtocol

A null-terminated string that names the negotiated security protocol, such as `"TLS 1.3"` or `"TLS 1.2"`. For plain HTTP and local content, this string is empty.

szCipher

A null-terminated string that identifies the selected cipher suite, such as `"TLS_AES_128_GCM_SHA256"`. This string is empty if the site is not secure or local content is being displayed.

szSubject

A null-terminated string which contains the certificate subject. Typically this includes the common name (CN) and/or subject alternative name which matches the hostname.

szIssuer

A null-terminated string which specifies the certificate issuer. This is normally the name of the Certificate Authority (CA) which provided the certificate.

ftValidFrom

The date and time when the certificate was issued. This is represented as a standard FILETIME structure.

ftValidTo

The date and time when the certificate expires. This is represented as a standard FILETIME structure.

Remarks

The security information reflects the state of the last navigation. During navigation, values can be provisional; the final, authoritative values are available after the navigation completes. The ***dwLevel*** value is an assessment that may be adjusted by the engine for conditions such as mixed content, weak ciphers, or certificate problems. The ***dwFlags*** field provides the underlying reasons and may have multiple bitflags set. Applications should check for combinations (for example, a page might be secure overall but also report a warning flag if subresources were blocked).

For plain HTTP (unencrypted) pages, ***dwLevel*** is typically `WEBVIEW_SECURITY_LEVEL_INSECURE`. The ***szProtocol*** and ***szCipher*** members are empty strings and ***dwFlags*** will include `WEBVIEW_SECURITY_FLAG_UNENCRYPTED`. For HTTPS with a valid certificate, ***dwLevel*** is typically `WEBVIEW_SECURITY_LEVEL_SECURE`, and other members identify the negotiated TLS version and cipher suite. If the certificate is expired or otherwise invalid, the security level is downgraded and the corresponding certificate flags are set.

The ***szSubject*** and ***szIssuer*** strings are for display and logging purposes; certificate parsing and validation is handled by the browser. The validity timestamps (***ftValidFrom*** and ***ftValidTo***) are provided to enable simple checks, but applications do not need to implement certificate validation logic themselves.

Set the ***dwSize*** member to the actual size of the structure before calling **GetSecurityInfo**. If the caller passes a different size than expected, the method will fail or only populate the fields known to that size.

Both ANSI (`A`) and Unicode (`W`) variants are provided. When compiling with `UNICODE` defined, the `TCHAR` strings are UTF-16. In ANSI builds, they are UTF-8. The **GetSecurityInfo** macro resolves to the appropriate version based on the project's character set.

Because the browser can update security state as additional resources are loaded, applications that display a security indicator should refresh it on navigation completion events. Treat the security level as ordered and do not "upgrade" a previously observed state during the same navigation unless a later callback provides a higher confidence result. Note that it's also possible for the security level to be `WEBVIEW_SECURITY_LEVEL_SECURE` during navigation, but may be downgraded to `WEBVIEW_SECURITY_LEVEL_WARNING` when navigation completes due to mixed content in the page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Unicode: Implemented as Unicode and ANSI versions

See Also

GetSecurityInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Constants

Name	Value	Description
Options		
WEBVIEW_OPTION_ENABLE_NOTHING	0	No options enabled. This is very restrictive and will not permit the use of JavaScript.
WEBVIEW_OPTION_ENABLE_ACCELERATORS	0x00001	Enables common keyboard accelerators (e.g., Ctrl+L, F5). Accelerator key events are surfaced via `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS`.
WEBVIEW_OPTION_ENABLE_CONTEXT_MENU	0x00002	Shows the built-in right-click context menu. Clear this to fully suppress the menu.
WEBVIEW_OPTION_ENABLE_STATUSBAR	0x00004	Displays status text (e.g., link targets, loading state) in the bottom-left status area.
WEBVIEW_OPTION_ENABLE_ZOOM_CONTROL	0x00008	Allows user zoom (Ctrl+Plus/Minus, Ctrl+mouse-wheel). If disabled, zoom remains fixed unless set by API.
WEBVIEW_OPTION_ENABLE_AUDIO	0x00010	Permits audio playback in the WebView. Disable to mute at the engine level.
WEBVIEW_OPTION_ENABLE_AUTOFILL	0x00020	Enables form autofill suggestions where available.
WEBVIEW_OPTION_ENABLE_PASSWORD_AUTOSAVE	0x00040	Allows the engine to offer saving website passwords/credentials.
WEBVIEW_OPTION_ENABLE_PRIVACY	0x00080	Enable private browsing
WEBVIEW_OPTION_ENABLE_SCRIPTING	0x00100	Enables JavaScript execution. Disabling blocks script evaluation and script-initiated behaviors.
WEBVIEW_OPTION_ENABLE_DIALOGS	0x00200	Allows script dialogs (alert/confirm/prompt) and related UI. Disable to suppress script-driven dialogs.
WEBVIEW_OPTION_ENABLE_DEVTOOLS	0x00400	Permits opening Developer Tools (e.g., F12). Disable to block DevTools access.
WEBVIEW_OPTION_ENABLE_NEW_WINDOWS	0x00800	Allows the creation of new popup windows
WEBVIEW_OPTION_ENABLE_MESSAGES	0x01000	Enables `window.chrome.webview.postMessage` communication from the page to the host.
WEBVIEW_OPTION_ENABLE_EXTERNAL_DROP	0x02000	Allows external drag-and-drop (e.g., files from Explorer) into the WebView surface.
WEBVIEW_OPTION_RESTRICT_NAVIGATION	0x10000	Blocks navigations after the initial page

		load unless explicitly permitted by the host.
WEBVIEW_OPTION_RESTRICT_DOWNLOADS	0x20000	Blocks attempts to download files through the browser.
WEBVIEW_OPTION_EXCLUSIVE_PROFILE	0x40000	Require exclusive use of the user data folder
WEBVIEW_OPTION_TRANSPARENT_BACKGROUND	0x80000	Renders with a transparent background where content allows; the host window should support composition.
WEBVIEW_OPTION_ENABLE_DEFAULT	0x0031F	Convenience preset enabling the most common interactive features for typical app scenarios.
Navigation Flags		
WEBVIEW_NAVIGATION_REDIRECTED	0x0001	The navigation was the result of an HTTP redirect.
WEBVIEW_NAVIGATION_USER_INITIATED	0x0002	Navigation began from a user gesture rather than being initiated by a script.
WEBVIEW_NAVIGATION_LOCAL_DATA	0x1000	Content was served using local data rather than a full network fetch.
Accelerator Key Flags		
WEBVIEW_ACCELERATOR_KEY_DOWN	0x0001	Indicates an accelerator key-down event.
WEBVIEW_ACCELERATOR_KEY_UP	0x0002	Indicates an accelerator key-up event.
WEBVIEW_ACCELERATOR_KEY_SYSTEM	0x0004	Indicates a system key event using the Alt key, similar to `WM_SYSKEY`.
History Movement Flags		
WEBVIEW_MOVE_BACKWARD_ENABLED	0x0001	Backward navigation is currently available.
WEBVIEW_MOVE_FORWARD_ENABLED	0x0002	Forward navigation is currently available.
Navigation Error Codes		
WEBVIEW_ERROR_NONE	0	No error.
WEBVIEW_ERROR_CERTIFICATE_COMMON_NAME	1	Certificate common name mismatch.
WEBVIEW_ERROR_CERTIFICATE_EXPIRED	2	Certificate is expired.
WEBVIEW_ERROR_INVALID_CLIENT_CERTIFICATE	3	Invalid or untrusted client certificate.
WEBVIEW_ERROR_CERTIFICATE_REVOKED	4	Certificate has been revoked.
WEBVIEW_ERROR_INVALID_CERTIFICATE	5	Certificate is invalid for another reason.
WEBVIEW_ERROR_SERVER_UNREACHABLE	6	Cannot reach the server/host.
WEBVIEW_ERROR_TIMEOUT	7	Operation timed out.
WEBVIEW_ERROR_INVALID_SERVER_RESPONSE	8	Server returned an invalid or unexpected response.

WEBVIEW_ERROR_CONNECTION_ABORTED	9	Connection aborted by the server or network.
WEBVIEW_ERROR_CONNECTION_RESET	10	Connection was reset.
WEBVIEW_ERROR_DISCONNECTED	11	Disconnected during the operation.
WEBVIEW_ERROR_CANNOT_CONNECT	12	Failed to establish a connection.
WEBVIEW_ERROR_HOST_NAME_NOT_RESOLVED	13	DNS resolution failed for the host.
WEBVIEW_ERROR_OPERATION_CANCELED	14	Operation was canceled by the user or host.
WEBVIEW_ERROR_REDIRECT_FAILED	15	Redirection failed.
WEBVIEW_ERROR_UNEXPECTED_ERROR	16	An unexpected error occurred.
WEBVIEW_ERROR_CREDENTIALS_REQUIRED	17	Authentication credentials are required.
WEBVIEW_ERROR_PROXY_AUTHENTICATION_REQUIRED	18	Proxy authentication is required.
Events		
WEBVIEW_EVENT_WINDOW_ATTACHED	1	attached to the host window. This is the first event that fires after the browser is created.
WEBVIEW_EVENT_WINDOW_DETACHED	2	detached from the host window. This is the last event that fires before the browser is destroyed.
WEBVIEW_EVENT_FOCUS_CHANGED	3	focus changed. The ***wParam*** value indicates if focus was gained (1) or lost (0).
WEBVIEW_EVENT_ACCELERATOR_KEYPRESS	4	Accelerator key event. The ***wParam*** value is the virtual-key code and ***lParam*** is the accelerator flags.
WEBVIEW_EVENT_NAVIGATION_STARTING	5	Navigation starting. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the navigation flags.
WEBVIEW_EVENT_NAVIGATION_COMPLETED	6	Navigation completed. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the error code.
WEBVIEW_EVENT_SECURITY_CHANGED	7	Security level changed. The ***wParam*** is the security level and ***lParam*** is the security flags.
WEBVIEW_EVENT_HISTORY_CHANGED	8	Back/forward availability changed. The ***wParam*** value is the history movement flags.
WEBVIEW_EVENT_DOWNLOAD_STARTING	9	A download has been initiated. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the file ID.
WEBVIEW_EVENT_DOWNLOAD_COMPLETED	10	A download has completed. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the file ID.

WEBVIEW_EVENT_MESSAGE_RECEIVED	11	Message received from a script on the page.
WEBVIEW_EVENT_SCRIPT_COMPLETED	12	Script execution completed. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the user-defined parameter.
Status Flags		
WEBVIEW_STATUS_SESSION_IDLE	0	No navigation in progress, the browser is in an idle state.
WEBVIEW_STATUS_NAVIGATING	0x0001	Currently navigating to a new location.
WEBVIEW_STATUS_NAVIGATION_ERROR	0x0002	The navigation failed due to an error.
WEBVIEW_STATUS_NAVIGATION_CANCELED	0x0004	The navigation was canceled by the user or host.
WEBVIEW_STATUS_NAVIGATION_TIMEOUT	0x0008	The navigation exceeded the configured timeout period.
WEBVIEW_STATUS_NAVIGATION_BLOCKED	0x0010	The navigation was blocked by policy.
WEBVIEW_STATUS_CONNECTION_FAILED	0x0100	The connection attempt failed or hostname was invalid.
WEBVIEW_STATUS_CONNECTION_ABORTED	0x0200	The connection was aborted or reset.
WEBVIEW_STATUS_SECURITY_WARNING	0x1000	The page includes insecure/mixed content or weak encryption was detected.
Icon Flags		
WEBVIEW_ICON_BIG	0x0001	Update the large window icon.
WEBVIEW_ICON_SMALL	0x0002	Update the small window icon.
WEBVIEW_ICON_BOTH	0x0003	Update both large and small icons.
WEBVIEW_ICON_DPI_AWARE	0x0100	Choose the system-recommended small icon size for the current DPI when possible.
Navigate Flags		
WEBVIEW_NAVIGATE_DEFAULT	0	Default navigation behavior.
WEBVIEW_NAVIGATE_SAFE_URL	0x0001	Applies additional URL safety checks and normalization before navigating.
Header Type Flags		
WEBVIEW_HEADERS_REQUEST	0x0001	Specifies custom request headers supplied by the client.
WEBVIEW_HEADERS_RESPONSE	0x0002	Specifies headers returned by the server in the last response.
Print Dialog Flags		

WEBVIEW_PRINT_DIALOG_DEFAULT	0	Use Edge's print preview dialog, equivalent to the Ctrl+P experience.
WEBVIEW_PRINT_DIALOG_SYSTEM	0x0001	Use the native system print dialog. Does not include print preview.
File Types (Save Document)		
WEBVIEW_FILE_TYPE_DEFAULT	0	Save using a format inferred from the file extension.
WEBVIEW_FILE_TYPE_TEXT	1	Save the document as a plain text file.
WEBVIEW_FILE_TYPE_HTML	2	Save the document as an HTML file.
WEBVIEW_FILE_TYPE_JSON	3	Save the document as JSON formatted string with reserved characters encoded.
WEBVIEW_FILE_TYPE_MHTML	4	Save the document as an MHTML single-file archive which includes all page assets.
WEBVIEW_FILE_TYPE_PDF	5	Save the document as a PDF file.
WEBVIEW_FILE_TYPE_JPEG	101	Save the current view as a JPEG image.
WEBVIEW_FILE_TYPE_PNG	102	Save the current view as a PNG image.
WEBVIEW_FILE_TYPE_WEBP	103	Save the current view as a WEBP image.
Download Status		
WEBVIEW_DOWNLOAD_STARTED	1	The file download has started and is in-progress.
WEBVIEW_DOWNLOAD_COMPLETED	2	The file download has completed successfully.
WEBVIEW_DOWNLOAD_CANCELED	3	The file download has been canceled.
WEBVIEW_DOWNLOAD_FAILED	4	The file download failed due to a network error or other problem.
Zoom		
WEBVIEW_ZOOM_MIN_PERCENT	25	Minimum zoom level as a percentage.
WEBVIEW_ZOOM_MAX_PERCENT	400	Maximum zoom level as a percentage.
WEBVIEW_ZOOM_DEFAULT	100	Default zoom level as a percentage.
Security Levels		
WEBVIEW_SECURITY_LEVEL_UNKNOWN	0	The security state could not be determined.
WEBVIEW_SECURITY_LEVEL_INSECURE	1	The page was retrieved using plain HTTP or the site has an invalid certificate.
WEBVIEW_SECURITY_LEVEL_WARNING	2	The page contains mixed content or only weak encryption is available.
WEBVIEW_SECURITY_LEVEL_NEUTRAL	3	Local content or the browser is in a transitional/loading state.

WEBVIEW_SECURITY_LEVEL_SECURE	4	The page was retrieved using HTTPS and the site has a valid certificate.
WEBVIEW_SECURITY_LEVEL_LOW	1	Alias of `WEBVIEW_SECURITY_LEVEL_WARNING`.
WEBVIEW_SECURITY_LEVEL_HIGH	4	Alias of `WEBVIEW_SECURITY_LEVEL_SECURE`.
Security Flags		
WEBVIEW_SECURITY_FLAG_NONE	0	No additional security indicators.
WEBVIEW_SECURITY_FLAG_UNENCRYPTED	0x0001	Content is not encrypted.
WEBVIEW_SECURITY_FLAG_INSECURE	0x0002	Site is insecure using outdated protocols or weak ciphers.
WEBVIEW_SECURITY_FLAG_CERT_WEAK	0x0004	The certificate uses weak signatures.
WEBVIEW_SECURITY_FLAG_CERT_ERROR	0x0008	The certificate is invalid. This is a general error condition.
WEBVIEW_SECURITY_FLAG_CERT_EXPIRED	0x0010	The certificate expired but is otherwise valid.
WEBVIEW_SECURITY_FLAG_UNSAFE	0x0020	The site is considered potentially unsafe (it failed reputation checks).

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Class Events

The class library reports state changes and notifications through a user-defined callback function which is specified when a browser instance is created or attached to a window. For more information, refer to the **WebViewEventProc** function. It is not required for an application to implement handlers for each event. When there is no event handler for a specific event, the browser takes its default action for that event. For example, if your application does not implement a handler for the `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS` event then all accelerator key events are processed normally by the browser control.

Event Constants

Name	Description
WEBVIEW_EVENT_WINDOW_ATTACHED	This event is raised after the WebView2 control has been successfully attached to the specified host window. At this point the browser environment has been created, the controller has been initialized, and the browser instance is ready to accept navigation requests and configuration changes. Applications often use this event to perform initialization tasks such as setting a custom user agent, applying option flags, or starting the first navigation to a default URL. The event confirms that the control is operational and that subsequent API calls will succeed. The *wParam* value indicates success (1) or failure (0). If the browser could not be attached to the window, the *lParam* value will contain the HRESULT code returned by the browser's COM interface.
WEBVIEW_EVENT_WINDOW_DETACHED	This event occurs when the WebView2 control has been detached from its host window, either because the host window is being destroyed or because the application explicitly requested detachment. After this event has fired, the instance is no longer valid and should not be used for navigation or script execution. Applications typically use this event to clean up resources associated with the WebView, save state, or update their user interface to reflect that the browser view is no longer available. Any references to the WebView instance should be considered invalid after detachment.
WEBVIEW_EVENT_FOCUS_CHANGED	This event is raised whenever the browser window gains or loses keyboard focus relative to its host window. The *wParam* value indicates if focus was gained (1) or lost (0). This allows the application to track when user input is directed to the embedded browser versus other controls in the window. For example, you might use this event to update menu state, adjust shortcut key handling, or provide visual cues when the browser window is active.
WEBVIEW_EVENT_ACCELERATOR_KEYPRESS	This event is raised when the user presses an accelerator key combination such as the arrow keys, Page Up and Page Down, Insert, Delete, or other keys that the embedded

	browser normally handles. The ***wParam*** value is the virtual-key code and ***lParam*** is the accelerator flags. The event gives your application a chance to intercept these keystrokes before they are processed by the browser. This is especially useful when you want to provide custom navigation, prevent certain keys from being used, or integrate the control with application-level keyboard shortcuts. To prevent the default behavior being taken by the browser, call the **KeyEventHandled** method from within this event.
WEBVIEW_EVENT_NAVIGATION_STARTING	This event is fired when a new navigation request begins. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the navigation flags. The event is raised before any network request is made, ensuring that no data is sent until the handler has had a chance to approve or deny the navigation. The **GetNavigationUrl** method can be called to determine the new URL the browser is navigating to and the **CancelNavigation** method can be used to prevent the navigation.
WEBVIEW_EVENT_NAVIGATION_COMPLETED	This event is raised when a navigation request has finished loading. It is triggered regardless of whether the navigation succeeded or failed, with the ***wParam*** value indicating success (1) or failure (0). If the navigation has failed, the ***lParam*** value is the error code. Applications typically handle this event to update their UI once the content has loaded, or to implement custom error handling when navigation fails. Because the event always fires, even on errors, it is a reliable notification point for the end of a navigation sequence.
WEBVIEW_EVENT_SECURITY_CHANGED	This event occurs when the security level of the current connection changes. The ***wParam*** value is the security level and ***lParam*** is the security flags. For example, the level may change from neutral to secure once a certificate has been validated, or from secure to warning if mixed content is detected. Additional information about the state of the connection can be obtained by calling the **GetSecurityInfo** method. Applications often use this event to update visual indicators such as a padlock icon or to apply additional security-related policies. It is important to note that this event may be raised multiple times during a single navigation, and earlier values should be discarded in favor of the most recent state.
WEBVIEW_EVENT_HISTORY_CHANGED	This event occurs whenever the browsing history changes. The ***wParam*** value is the history movement flags. Navigating to a new page, using the **GoBack** or **GoForward** methods, or programmatically calling navigation methods can trigger this event. Your application can respond by enabling or disabling UI elements such as toolbar buttons depending on whether it is possible to go

	back or forward.
WEBVIEW_EVENT_DOWNLOAD_STARTING	This event occurs when a file download is initiated either by a user clicking on a link or navigating to a URL which specifies content that cannot be rendered in the browser. The *wParam* value indicates success (1) or failure (0) and *lParam* is the unique file identifier which can be used to identify the specific download. This identifier can be used with the **GetDownloadInfo** method to obtain information about the download in progress.
WEBVIEW_EVENT_DOWNLOAD_COMPLETED	This event occurs when a file download has completed. The *wParam* value indicates success (1) or failure (0) and *lParam* is the unique file identifier. If the file download is canceled, it is considered to be a failed download. The **GetDownloadInfo** method can be used to obtain information about the download, such as the total number of bytes copied, how much time elapsed and the full path of the local file that was created.
WEBVIEW_EVENT_MESSAGE_RECEIVED	This event occurs when a message is posted from JavaScript code in the web page using `window.chrome.webview.postMessage`. The message is delivered as a string, which can be retrieved using the **GetWebMessage** method. The message may include structured data such as JSON or it may be a simple JSON formatted string. This event provides a two-way communication channel between the page content and the host application, allowing developers to implement hybrid scenarios where the web page can notify the native host of user actions, requests, or state changes. If messages are not enabled, this event will not be raised.
WEBVIEW_EVENT_SCRIPT_COMPLETED	This event is raised when a script execution initiated by the application completes. The *wParam* value indicates success (1) or failure (0) and *lParam* is the user-defined parameter provided when the **AsyncExecute** method was called. The **GetWebResult** method can be called to obtain the result of the script execution. If the script contained errors or could not be executed, the status will reflect a failure condition. Applications typically use this event to retrieve values from the page, execute custom logic in response to script results, or detect when asynchronous script operations have finished. Because script execution is asynchronous, this event is the definitive way to know when a script call has finished.

Remarks

Some events, such as `WEBVIEW_EVENT_NAVIGATION_STARTING` and `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS`, allow your application to influence behavior by canceling an action or handling a keypress. Others, like `WEBVIEW_EVENT_HISTORY_CHANGED` or `WEBVIEW_EVENT_SECURITY_CHANGED`, are informational and allow you to update user interface

elements or record state.

If your application does not enable specific options (such as allowing scripts, dialogs, or message passing), the corresponding events will not be raised. For example, if messages are disabled, the `WEBVIEW_EVENT_MESSAGE_RECEIVED` event will never occur. This makes it important to configure options appropriately for the scenarios you wish to support.

See Also

Attach, Create, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Class Options

When creating or attaching a WebView instance, a set of options can be specified that control its behavior. These options are defined as a collection of bitmask flags that can be combined together. The *dwOptions* parameter passed to methods such as **Create** or **Attach** may include one or more of these values.

Option Constants

Name	Description
WEBVIEW_OPTION_ENABLE_ACCELERATORS	Enables support for browser accelerator keys such as function keys (F1 for help, F5 for refresh, etc.) and standard navigation shortcuts. With this option enabled, the WebView responds to built-in key combinations in the same way a full browser does. Disabling it prevents end users from triggering those behaviors, which may be desirable in kiosk or custom application scenarios where you want to restrict keyboard input to only what your application explicitly supports.
WEBVIEW_OPTION_ENABLE_CONTEXT_MENU	Enables the default right-click context menu provided by the browser. This menu exposes commands such as Back, Forward, Reload, and View Source. In an embedded control, this may not always be appropriate since it can expose functionality you do not want available to users. Disabling this option suppresses the default context menu entirely, leaving you free to implement your own context handling if needed.
WEBVIEW_OPTION_ENABLE_STATUSBAR	Shows the built-in status bar, which normally appears at the bottom left of the control when hovering over a link. It provides basic feedback about navigation targets but has no interactive features. In a minimal UI, you may prefer to disable this so the embedded control presents only the rendered page without browser chrome. Developers can still query navigation state programmatically regardless of this setting.
WEBVIEW_OPTION_ENABLE_ZOOM_CONTROL	Enables zoom support through keyboard and mouse input (Ctrl+Plus, Ctrl+Minus, Ctrl+Mouse Wheel). With this enabled, end users can adjust the document's scaling factor directly. When disabled, zooming is locked at 100%, ensuring that the page is always displayed at its natural size. This can be useful in applications where layout fidelity is important and user scaling could cause unexpected clipping or misalignment.
WEBVIEW_OPTION_ENABLE_AUDIO	Enables audio playback within the WebView. When this option is disabled, the control is effectively

	muted and no sound is produced, regardless of the page content. Applications that need to enforce silent operation, such as monitoring dashboards, may want to leave this option disabled. This flag maps to the runtime's muting APIs and can be toggled dynamically after creation.
WEBVIEW_OPTION_ENABLE_AUTOFILL	Enables the use of autofill for HTML forms. When enabled, the control can suggest previously entered values for form fields such as names, addresses, or other personal data. Disabling this option prevents the runtime from remembering or suggesting values, which is often preferable in scenarios where privacy is a concern or where data entry should always be manual.
WEBVIEW_OPTION_ENABLE_PASSWORD_AUTOSAVE	Enables automatic saving of credentials in the runtime's password store. When this option is set, the control prompts to save usernames and passwords entered in forms, making them available for later autofill. While convenient for end users, applications embedding the control may choose to disable this to avoid storing sensitive information on the client system, particularly in enterprise or shared environments.
WEBVIEW_OPTION_ENABLE_PRIVACY	Enables the equivalent of InPrivate browsing in Microsoft Edge or Incognito mode in Google Chrome. If this option is specified, the user profile is not persisted beyond the current session. Browsing and search history will not be preserved, all cookies are temporary, data entered in form fields will not be saved and passwords will not be stored. Any general location information used by the browser is cleared when the instance is closed. This option can only be specified when the browser instance is created or attached to a window. You cannot enable privacy mode after a browser session has started.
WEBVIEW_OPTION_ENABLE_SCRIPTING	Enables JavaScript execution within the WebView. Nearly all modern websites depend on JavaScript, so disabling this option will significantly limit what pages can display or function correctly. However, for controlled content where scripting is unnecessary or potentially unsafe, disabling this can enforce a higher level of security and prevent active content from running. Some other options, such as enabling dialogs shown in the browser or receiving messages from the web application requires scripting to be enabled. If your application is only displaying local, static HTML content then you may not need to enable this option.

WEBVIEW_OPTION_ENABLE_DIALOGS	Enables support for default script dialogs such as alert, confirm, and prompt. If disabled, those dialogs are suppressed and scripts attempting to display them will fail silently. This prevents unexpected modal UI from appearing in your application, but may also break sites that rely on these dialogs for basic interaction. If you enable scripting, it is recommended that you enable this option as well.
WEBVIEW_OPTION_ENABLE_DEVTOOLS	Enables the integrated Developer Tools (DevTools) interface. When enabled, users can press standard shortcuts (such as F12) or use the context menu to open the DevTools window and inspect the page. For production applications, you may want to disable this to prevent end users from inspecting or modifying content. During development or debugging, enabling it provides the same diagnostic features found in Microsoft Edge.
WEBVIEW_OPTION_ENABLE_NEW_WINDOWS	Enables the creation of new popup windows by scripts and/or support the creation of new browser windows (for example, if a link target is "_blank"). By default, navigations that would open a new window are redirected internally so the new page is displayed within the same browser window. Note that if you permit the creation of new popup windows, it can shift focus away from your main browser window. There is no mechanism in the API to enumerate the child popup windows and they are independent of the window attached to the browser.
WEBVIEW_OPTION_ENABLE_MESSAGES	Enables communication between the web page and the host application through the **`window.chrome.webview.postMessage`** API. This allows JavaScript running in the page to send messages back to your application, enabling advanced integration scenarios. If disabled, such messages are blocked and the corresponding events will never occur in your host code. This is commonly enabled when embedding custom HTML user interfaces that need to exchange data with the native application.
WEBVIEW_OPTION_ENABLE_EXTERNAL_DROP	Enables the WebView to accept external file drops, such as dragging and dropping a file from Windows Explorer into the control. If disabled, drop operations are ignored and the page cannot receive file input in this way. Applications that want to support drag-and-drop uploads or similar features can enable this option, but in contexts where security is important it may be best to disable it to prevent accidental or malicious file submission.

	Accepting dropped files requires scripting to be enabled.
WEBVIEW_OPTION_RESTRICT_NAVIGATION	Restricts navigation after the initial page has loaded. Attempts to navigate beyond this initial page will automatically be canceled when this option is specified. This is useful for scenarios like embedded help viewers, dashboards, or authentication windows where you want to confine the user to a specific site or document. The restriction applies to both user-initiated actions such as clicking links and programmatically navigating to a new URL.
WEBVIEW_OPTION_RESTRICT_DOWNLOADS	Prevents the browser from downloading files. Attempts to download a file will be immediately canceled and no download UI will be displayed to the user. Normally, downloads are permitted and files will be stored in the current user's Downloads folder. If you wish to selectively prevent certain types of downloads, you can monitor the download events and decide individually whether the file should be downloaded based on the name, extension or location.
WEBVIEW_OPTION_EXCLUSIVE_PROFILE	Requires exclusive use of the default user profile stored in the data folder selected when the browser instances is created. If this option is enabled, the browser will attempt to obtain an exclusive lock on the profile which stores the current user's cached pages, browser history, cookies and passwords. Normally this session data can be shared with other processes, but this option ensures that only the current process may access it while the browser session is active. If another browser session is sharing the same data, the creation process will fail. This option can only be specified when the browser instance is created or attached to a window.
WEBVIEW_OPTION_TRANSPARENT_BACKGROUND	Configures the WebView to render with a transparent background rather than an opaque surface. This allows the host application to blend the browser content with custom UI elements or backgrounds. It is particularly useful for overlay scenarios, custom skins, or when compositing content with layered windows. Developers should be aware that transparent rendering may affect performance on some systems and should test carefully in their application's UI context.

Remarks

Most of these options can be toggled dynamically at runtime using the **SetOption** method, although some are only effective when set at creation time. Options are applied by mapping directly to the

underlying WebView2 runtime's settings and may depend on a minimum runtime version. If an option is not supported by the installed version of the WebView2 runtime, calls to enable it will fail silently.

The **WEBVIEW_OPTION_ENABLE_DEFAULT** macro combines the most commonly used options, including accelerator keys, context menu, status bar, zoom control, audio, scripting, and dialogs. This provides a browser-like experience suitable for most applications. Developers with stricter requirements for security or a highly customized UI should select individual flags instead of relying on the defaults. For applications in managed environments, kiosk systems, or where privacy is a concern, it is recommended to only enable those specific features which are required for normal operation.

The **WEBVIEW_OPTION_ENABLE_PRIVACY** option causes the WebView2 instance to create a unique, temporary user data folder with a new user profile. It will not have access to the browsing history, saved form data or passwords for any other instance. When the browser is closed or detached from its host window, that temporary profile will be automatically deleted. There can be a minor performance cost related to creating and deleting these temporary profiles, particularly if a large amount of data has been cached during the session.

As with private browsing using a web browser like Microsoft Edge or Google Chrome, enabling privacy does not prevent an employer or Internet provider from being able to monitor your activity. Websites can still track your behavior, particularly if scripting is enabled because most sites use scripts as part of their analytics tracking. Private browsing will block third-party cookies by default, but otherwise permits the use of cookies during the session. Any data stored using the HTML5 web storage APIs will also be deleted when a private session ends.

See Also

Attach, Create, GetAllOptions, GetOption, SetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Whois Protocol Class Library

Request registration information for an Internet domain name.

Reference

- Class Methods
- Error Codes

Library Information

Class Name	CWhoisClient
File Name	CSWHOV12.DLL
Version	12.0.3024.1286
LibID	A3CFB881-EB11-4A10-95A2-C357BF955F74
Import Library	CSWHOV12.LIB
Dependencies	None
Standards	RFC 954

Overview

The Whois protocol library provides an interface for requesting registration information for an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The library provides an API for requesting that information and returning it to the program so that it can be displayed or processed. This library would be most commonly used to query the server at **whois.internic.net** to obtain information about a specific Internet domain name or an administrative contact at that domain.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing

applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Whois Protocol Class Methods

Class	Description
CWhoisClient	Constructor which initializes the current instance of the class
~CWhoisClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
Connect	Connect to the specified server
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of network function calls to a file
FreezeEvents	Suspend or resume event handling by the calling process
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetStatus	Return the current client status.
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if there is data available to be read from the server
Read	Read data returned by the server
RegisterEvent	Register an event callback function
Search	Search for the specified record
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
WhoisEventProc	Callback method that processes events generated by the client

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::CWhoisClient Method

```
CWhoisClient();
```

The **CWhoisClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **WhoisInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

~CWhoisClient, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::~CWhoisClient

```
~CWhoisClient();
```

The **CWhoisClient** destructor releases resources allocated by the current instance of the **CWhoisClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CWhoisClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CWhoisClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::AttachHandle Method

```
VOID AttachHandle(
   HCLIENT hClient
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the ***hClient*** parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

AttachThread, DetachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::AttachThread Method

```
DWORD AttachThread(
   DWORD dwThreadId
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

AttachHandle, Cancel, Connect, DetachHandle, Disconnect, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

IsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::Connect Method

```
BOOL Connect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 43.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Disconnect, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

AttachHandle, GetHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

EnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::Disconnect Method

```
VOID Disconnect();
```

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

Connect, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::EnableEvents Method

```
INT EnableEvents(
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation.

	This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

DisableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::EnableTrace Method

```
BOOL EnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::FreezeEvents Method

```
INT FreezeEvents(
   BOOL bFreeze
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DisableEvents, EnableEvents, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::GetErrorString Method

```
INT GetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);

INT GetErrorString(
   DWORD dwErrorCode,
   CString& strDescription
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetLastError, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value INVALID_CLIENT will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

AttachHandle, DetachHandle, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::GetLastError Method

```
DWORD GetLastError();

DWORD GetLastError(
   CString& strDescription
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError**(0) when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

GetErrorString, SetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
WHOIS_STATUS_UNUSED	No connection has been established.
WHOIS_STATUS_IDLE	The client is current idle and not sending or receiving data.
WHOIS_STATUS_CONNECT	The client is establishing a connection with the server.
WHOIS_STATUS_READ	The client is reading data from the server.
WHOIS_STATUS_WRITE	The client is writing data to the server.
WHOIS_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

IsBlocking, IsInitialized, IsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

Connect, IsReadable, Read, SetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Cancel, GetStatus, IsConnected, IsInitialized, IsReadable, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

GetStatus, IsBlocking, IsInitialized, IsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

CWhoisClient, IsBlocking, IsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::IsReadable Method

```
BOOL IsReadable(
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

GetStatus, IsBlocking, IsConnected, IsInitialized

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::Read Method

```
INT Read(
   LPBYTE lpBuffer,
   INT cbBuffer
);

INT Read(
   CString& strBuffer,
   INT cbBuffer
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

EnableEvents, IsBlocking, IsReadable, RegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::RegisterEvent Method

```
INT RegisterEvent(
   UINT nEventId,
   WHOISEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WhoisEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR.

To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **WhoisEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

DisableEvents, EnableEvents, FreezeEvents, WhoisEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::Search Method

```
INT Search(
   LPCTSTR lpszKeyword,
   UINT nSearchType
);
```

The **Search** method submits the specified keyword to the server.

Parameters

lpszKeyword

Points to a string which specifies the query keyword. It may be a handle, name or mailbox.

nSearchType

The type of search being performed. One of the following values may be used:

Value	Description
WHOIS_SEARCH_ANY	Search for any record that matches the specified keyword.
WHOIS_SEARCH_HANDLE	Search only for handles that match the specified keyword.
WHOIS_SEARCH_MAILBOX	Search only for mailboxes that match the specified keyword.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Example

```
CWhoisClient whoisClient;

// Connect to the WHOIS server at whois.internic.net
if (whoisClient.Connect(_T("whois.internic.net")) == FALSE)
{
    whoisClient.ShowError();
    return;
}

// Request information about the sockettools.com domain name
if (whoisClient.Search(_T("sockettools.com")) == WHOIS_ERROR)
{
    whoisClient.ShowError();
    whoisClient.Disconnect();
    return;
}

// Read the response from the server, which contains information
// about the domain and the registry that maintains the record
CString strResult;
CString strBuffer;
INT nResult;

do
{
    if ((nResult = whoisClient.Read(strBuffer)) > 0)
        strResult += strBuffer;
```

```
}
while (nResult > 0);

// If there was an error reading the data, alert the user
if (nResult == WHOIS_ERROR)
    whoisClient.ShowError();

whoisClient.Disconnect();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

Connect, Disconnect, IsReadable, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::SetLastError Method

```
VOID SetLastError(
   DWORD dwErrorCode
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

GetErrorString, GetLastError, ShowError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::SetTimeout Method

```
INT SetTimeout(
   UINT nTimeout
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

Connect, GetTimeout, IsReadable, Read

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWhoisClient::ShowError Method

```
INT ShowError(
   LPCTSTR lpszAppTitle,
   UINT uType,
   DWORD dwErrorCode
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetErrorString, GetLastError, SetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

- Domain Name Service Library
- File Encoding Library
- File Transfer Protocol Library
- File Transfer Server Library
- Hypertext Transfer Protocol Library
- Hypertext Transfer Server Library
- Internet Control Message Protocol Library
- Internet Message Access Protocol Library
- Large Language Model Library
- Mail Message Library
- Network News Protocol Library
- Network Time Protocol Library
- News Feed Library
- Post Office Protocol Library
- Remote Command Library
- Simple Mail Transfer Protocol Library
- Secure Shell Protocol Library
- SocketWrench Library
- Telnet Protocol Library
- Terminal Emulation Library
- Text Message Library
- Web Services Library
- WebView Browser Library
- Whois Protocol Library

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Library

Resolve domain names into Internet addresses and return information about a remote host, such as the servers that are responsible for accepting mail for the domain.

Reference

- Functions
- Constants
- Data Structures
- Error Codes

Library Information

File Name	CSDNSV12.DLL
Version	12.0.3024.1286
LibID	C887CFA2-F7A7-4A9F-8AF2-48ED79642023
Import Library	CSDNSV12.LIB
Dependencies	None
Standards	RFC 1034

Overview

The Domain Name Services (DNS) protocol is what applications use to resolve domain names into Internet addresses as well as provide other information about a domain. All of the SocketTools libraries provide basic domain name resolution functionality, but the Domain Name Services library gives an application direct control over what servers are queried, the amount of time spent waiting for a response and the type of information that is returned.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing

applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Functions

Function	Description
DnsAttachThread	Attach the specified client handle to another thread.
DnsCancel	Cancel an outstanding nameserver query.
DnsCloseHandle	Close the current client session.
DnsCreateHandle	Create a new handle for the current client session.
DnsDisableTrace	Disable logging of network function calls.
DnsEnableTrace	Enable logging of network function calls to a file.
DnsEnumHostAliases	Enumerate the aliases for the specified host name or address.
DnsEnumMailExchanges	Return a list of mail exchanges for the specified host name or IP address.
DnsFormatAddress	Convert a numeric IPv4 address to a string.
DnsGetAddress	Convert an IPv4 address string in dotted notation to a numeric IP address.
DnsGetAddressFamily	Return the address family for the specified IP address.
DnsGetDefaultHostFile	Return the full path to the default host file on the local system.
DnsGetDefaultServicesFile	Return the full path the default services file on the local system.
DnsGetErrorString	Return a description for the specified error code.
DnsGetHostAddress	Return the IP address for the specified hostname.
DnsGetHostByAddress	Return a pointer to data for the specified host IP address.
DnsGetHostByName	Return a pointer to data for the specified host name.
DnsGetHostFile	Return the name of the current host file.
DnsGetHostInfo	Return additional information for the specified host.
DnsGetHostName	Return the host name for the specified IP address.
DnsGetHostServices	Return a list of services supported by the specified host.
DnsGetLastError	Return the last error code.
DnsGetLocalAddress	Return the IP address for the local host.
DnsGetLocalDomain	Return the local domain name for the current session.
DnsGetLocalName	Return the local host name.
DnsGetMailExchange	Return the host that processes mail for the specified domain.
DnsGetRecord	Return record data for the current host.
DnsGetResolverAddress	Return address of last nameserver that resolved query.
DnsGetResolverOptions	Return the current resolver options for the specified client session.
DnsGetRetryCount	Get the number of times the query is sent to each server.
DnsGetServerAddress	Return the address of the specified nameserver.
DnsGetServerPort	Return the port of the specified nameserver.
DnsGetServiceName	Return the name of a service assigned to the specified port number.

DnsGetServicePort	Return the port number assigned to the specified service name.
DnsGetTimeout	Get the number of seconds until a query times out.
DnsHostNameToUnicode	Converts the canonical form of a host name to its Unicode version.
DnsInitialize	Initialize the library and validates the specified license key at runtime.
DnsMatchHostName	Match a host name against of list of addresses including wildcards.
DnsNormalizeHostName	Return the canonical form of a host name.
DnsRegisterServer	Add a nameserver address to the current session.
DnsReset	Reset the current client state.
DnsSetHostFile	Specify the name of an alternate file to use when resolving hostnames and IP addresses.
DnsSetLastError	Set the last error code.
DnsSetLocalDomain	Set the local domain name for the current session.
DnsSetResolverOptions	Set the resolver options for the specified client session.
DnsSetRetryCount	Set the number of times the query is sent to each server.
DnsSetTimeout	Set the number of seconds until a query times out.
DnsUninitialize	Terminates the use of the library.
DnsUnregisterServer	Remove a nameserver address from the current session.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsAttachThread Function

```
DWORD WINAPI DnsAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **DnsAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **DnsAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **DnsAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **DnsCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **DnsUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsCancel, DnsCloseHandle, DnsCreateHandle, DnsUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsCancel Function

```
INT WINAPI DnsCancel(
   HCLIENT hClient
);
```

The **DnsCancel** function cancels any outstanding queries initiated by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetHostByAddress, DnsGetHostByName, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsCloseHandle Function

```
INT WINAPI DnsCloseHandle(
   HCLIENT hClient
);
```

The **DnsCloseHandle** function closes the handle and releases any memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsCreateHandle, DnsUninitialize, DnsUnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsCreateHandle Function

```
HCLIENT WINAPI DnsCreateHandle(
    INT nTimeout,
    INT nRetries,
    DWORD dwOptions
);
```

The **DnsCreateHandle** function creates a new handle which can be used to query the default nameservers configured for the local host.

Parameters

nTimeout

The number of seconds that the client will wait for a response before failing the query. The constant DNS_TIMEOUT can be used to specify the default timeout period of 15 seconds.

nRetries

The number of attempts the client will make to resolve a query after a timeout. The constant DNS_RETRIES can be used to specify the default retry count of 4.

dwOptions

An unsigned integer that specifies one or more resolver options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
DNS_OPTION_NONE	No additional resolver options specified. This is the default value, and it is recommended that most applications do not specify additional options unless the implications of doing so are understood.
DNS_OPTION_PROMOTE	Promotes the server that successfully completed the last query to the first server that will be used to resolve subsequent queries. This option can improve performance in some cases where one or more of the registered servers are non-responsive. This option takes precedence over the DNS_OPTION_ROTATE option.
DNS_OPTION_ROTATE	Enables a round-robin selection of nameservers when performing queries. Normally each nameserver is queried in the same order. This option rotates the available nameservers so a different server is used with each query.
DNS_OPTION_AUTHONLY	Require the answer from the nameserver to be authoritative, not from the server's cache. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_PRIMARY	Queries are only accepted from the primary nameserver. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_NORECURSE	Disable the sending of recursive queries to the nameserver. Specifying this option will disable the bit in

	the DNS request header that specifies recursion is desired.
DNS_OPTION_NOSEARCH	Disable additional queries of higher domains in the search list if the host name cannot be resolved. If this option is specified, and the host name cannot be resolved using the local domain name an error is returned immediately. This option is ignored if no local domain has been specified or if the DNS_OPTION_NOSUFFIX option has been specified.
DNS_OPTION_NOSUFFIX	Disable additional queries using the local domain name if the host name is not a fully qualified domain name and cannot be resolved. This option is ignored if no local domain has been specified.
DNS_OPTION_NONAMECHECK	Disable checking the host name for invalid characters, such as the underscore and control characters. By default, host names are checked to ensure they're valid before submitting a query to the nameserver.
DNS_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsCreateHandle** function returns a handle to the application which can be used to perform nameserver queries. By default, the library will use the nameservers which are configured for the local host. To determine the IP addresses of those nameservers, use the **DnsGetServerAddress** function.

To specify alternate nameservers, either as replacements for or additions to the default nameservers, use the **DnsRegisterServer** function.

It is recommended that most applications do not specify any resolver options and use the default behavior. Specifying these options without understanding how they can affect standard queries can result in unexpected failures. In particular, caution should be used when specifying the DNS_OPTION_NORECURSE and DNS_OPTION_NOSEARCH options as they change the normal process of resolving a host name.

Specifying the DNS_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsCloseHandle, DnsGetResolverOptions, DnsGetServerAddress, DnsInitialize, DnsRegisterServer, DnsSetLocalDomain, DnsSetResolverOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsDisableTrace Function

```
BOOL WINAPI DnsDisableTrace();
```

The **DnsDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an internal failure to release the resources allocated for the logging session.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsEnableTrace Function

```
BOOL WINAPI DnsEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **DnsEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsEnumHostAliases Function

```
INT WINAPI DnsEnumHostAliases(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   LPTSTR *lpszHostAlias,
   INT nMaxAliases
);
```

The **DnsEnumHostAliases** function returns a list of aliases for the specified host name or IP address.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used. This value may also be an IP address.

lpszHostAlias

Pointer to an array of string pointers which specify one or more host aliases. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved.

nMaxAliases

The maximum number of aliases in the array. This parameter must have a value of at least one, or an error will be returned.

Return Value

If the function succeeds, the return value is the number of host aliases. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify the host aliases or free any of the values. This function uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsGetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsEnumMailExchanges Function

```
INT WINAPI DnsEnumMailExchanges(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   LPTSTR *lpszMailExchanges,
   INT nMaxMailExchanges
);
```

The **DnsEnumMailExchanges** function returns a list of mail exchanges for the specified host name or IP address.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string that contains the hostname or domain name to be queried.

lpszMailExchanges

Pointer to an array of string pointers which specify one or more mail exchanges. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved. The list of mail exchange records is sorted in priority order, from highest (i.e., those whose preference value is smallest) to lowest.

nMaxMailExchanges

The maximum number of mail exchanges in the array. If this parameter is 0, then the function will return the number of mail exchanges, but the list of mail exchanges will not be output in *lpszMailExchanges*.

Return Value

If the function succeeds, the return value is the number of mail exchanges. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify the mail exchange host names or free any of the values. This function uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other function calls.

Example

```
// Count the number of mail exchanges
if ((nMX = DnsEnumMailExchanges(hClient,szHostName, NULL, 0)) == DNS_ERROR)
{
    dwError = DnsGetLastError();
    DnsGetErrorString(dwError, szError, BUFSIZE);
    printf("DnsEnumMailExchanges failed for %s: %s\n", szHostName, szError);
}
else
{
    int nIndex;
    printf("%d MX for %s\n",nMX, szHostName);
```

```
    // Allocate memory for the list of mail exchanges
    lpszMailExchanges = (LPTSTR *)malloc(nMX * sizeof(LPTSTR));

    // Retrieve the list of mail exchanges
    nMX = DnsEnumMailExchanges(hClient, szHostName, lpszMailExchanges, nMX);

    for (nIndex = 0; nIndex < nMX; nIndex++)
         printf("#%d: %s\n", nIndex+1, *lpszMailExchanges++);

    free(lpszMailExchanges);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetMailExchange

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsFormatAddress Function

```
INT WINAPI DnsFormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **DnsFormatAddress** function converts a numeric IPv4 or IPv6 address to a string.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IPv4 or IPv6 address to be converted into a string.

lpszAddress

A null-terminated string which will contain the converted IPv4 address in dot-notation. This string should be at least 16 characters in length.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the function succeeds, the return value is the length of the string buffer. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetAddress, DnsGetHostAddress, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetAddress Function

```
INT WINAPI DnsGetAddress(
   LPCTSTR lpszAddress,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **DnsGetAddress** function converts an address string to a numeric IPv4 or IPv6 address.

Parameters

lpszAddress

A null-terminated string which specifies an IPv4 address in dotted notation.

nAddressFamily

An integer value which specifies the type of IP address. If this parameter is zero, the address family will be determined automatically based on the format of the address string. If this parameter is DNS_ADDRESS_IPV4, the address must be in IPv4 format, and if it is DNS_ADDRESS_IPV6, the address must be in IPv6 format.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the numeric form of the IPv4 or IPv6 address in network byte order when the function returns.

Return Value

If the function succeeds, the return value is the address family for the IP address. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

This function will only accept a string that is in the proper format for an IP address, and cannot be used to resolve a host name. To perform host name resolution, use the **DnsGetHostAddress** function. To convert a numeric address to an address string, use the **DnsFormatAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsFormatAddress, DnsGetHostAddress, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetAddressFamily Function

```
INT WINAPI DnsGetAddressFamily(
   LPCTSTR lpszAddress
);
```

The **DnsGetAddressFamily** function returns the address family for the specified IP address.

Parameters

lpszAddress

A null-terminated string which specifies an IPv4 or IPv6 address.

Return Value

If the function succeeds, the return value is DNS_ADDRESS_IPV4 if the address is in IPv4 format, or DNS_ADDRESS_IPV6 if the address is in IPv6 format. If the address string is not in a recognized format, it returns DNS_ADDRESS_ANY.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsFormatAddress, DnsGetHostAddress, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetDefaultHostFile Function

```
INT WINAPI DnsGetDefaultHostFile(
    LPTSTR lpszFileName,
    INT nMaxLength
);
```

The **DnsGetDefaultHostFile** function returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **DnsGetHostAddress** and **DnsGetHostName** functions. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified path to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **DnsGetLastError**.

Remarks

This function returns the location of the host file and does not determine if the file actually exists. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname. To specify an alternate local host file, use the **DnsSetHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetDefaultServicesFile, DnsGetHostAddress, DnsGetHostFile, DnsGetHostName, DnsSetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetDefaultServicesFile Function

```
INT WINAPI DnsGetDefaultServicesFile(
    LPTSTR lpszFileName,
    INT nMaxLength
);
```

The **DnsGetDefaultServicesFile** function returns the fully qualified path name of the services file on the local system. The services file is used as a database that maps a port number to a service name, and is used by the **DnsGetServiceName** and **DnsGetServicePort** functions. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
name port/protocol [alias ...]
```

For example, one typical entry maps the service name "http" to port 80, the standard port for the Hypertext Transfer Protocol. This would be entered as:

```
http 80/tcp www www-http
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified path to the default services file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. A return value of zero indicates that the default services file could not be determined for the current platform. To get extended error information, call **DnsGetLastError**.

Remarks

This function returns the location of the services file and does not determine if the file actually exists. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetDefaultHostFile, DnsGetServiceName, DnsGetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetErrorString Function

```
INT WINAPI DnsGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **DnsGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **DnsGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetLastError, DnsSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostAddress Function

```
INT WINAPI DnsGetHostAddress(
   HCLIENT hClient,
   INT nAddressFamily,
   LPCTSTR lpszHostName,
   LPTSTR lpszHostAddress,
   INT nMaxLength
);
```

The **DnsGetHostAddress** function resolves the specified host name, storing the IP address in the provided buffer.

Parameters

hClient

Handle to the client session.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of DNS_ADDRESS_IPV4 specifies that the IPv4 address for the host should be returned. A value of DNS_ADDRESS_IPV6 specifies that the IPv6 address for the host should be returned. A value of DNS_ADDRESS_ANY specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host.

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

lpszHostAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 48 characters in length to accommodate both IPv4 and IPv6 addresses. The format of the address is determined by the address family specified.

nMaxLength

The maximum length of the string buffer. The maximum length of the buffer must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the host address buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetHostAddress** function may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the ***nAddressFamily*** parameter as DNS_ADDRESS_IPV4 to prevent the possibility of an IPv6 address being returned.

If the ***nAddressFamily*** parameter is specified as DNS_ADDRESS_ANY, this function will first check for an IPv4 address record for the host. If it exists, it will return that address. If the host does not have an IPv4 address, it will then check for an IPv6 address record and return that address. This gives preference to IPv4 addresses, but your application should never depend on this behavior. In the future, this function may change to give preference to IPv6 addresses.

To determine what format an address is in, use the **DnsGetAddressFamily** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsEnumHostAliases, DnsGetAddressFamily, DnsGetHostName, DnsGetMailExchange, DnsGetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostByAddress Function

```
LPHOSTENT WINAPI DnsGetHostByAddress(
    HCLIENT hClient,
    LPVOID lpvAddress,
    INT cbAddress,
    INT nAddressFamily
);
```

The **DnsGetHostByAddress** function returns a pointer to a HOSTENT structure which contains the results of a successful search for the host specified by address parameter.

Parameters

hClient

Handle to the client session.

lpvAddress

Pointer to an integer IPv4 address in network byte order.

cbAddress

The length of the address in bytes; this value should always be 4.

nAddressFamily

The type of address being resolved; this value should always be DNS_ADDRESS_IPV4 as defined in the Windows Sockets header file.

Return Value

If the function succeeds, the return value is a pointer to a HOSTENT structure. If the function fails, the return value is NULL. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. To convert an IPv4 address string in dotted notation to a 32-bit IP address, use the **DnsGetAddress** function.

This function is included for compatibility with existing applications which already use the HOSTENT structure. Because this function returns a pointer to a complex structure, it may not be suitable for some programming languages. There is no Unicode version of the HOSTENT structure and members will return pointers to ANSI strings even if the application is compiled to use Unicode.

This function is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **DnsGetHostAddress** and **DnsGetHostName** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsGetHostByName, DnsGetHostInfo, DnsGetHostName, DnsGetHostServices,

DnsGetRecord, DnsGetResolverAddress, DnsRegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostByName Function

```
LPHOSTENT WINAPI DnsGetHostByName(
    HCLIENT hClient,
    LPCTSTR lpszHostName
);
```

The **DnsGetHostByName** function returns a pointer to a HOSTENT structure which contains the results of a successful search for the host specified in the name parameter.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

Return Value

If the function succeeds, the return value is a pointer to a HOSTENT structure. If the function fails, the return value is NULL. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. This function will automatically resolve an IP address passed as a string, converting it to numeric form and calling the **DnsGetHostByAddress** function.

This function is included for compatibility with existing applications which already use the HOSTENT structure. Because this function returns a pointer to a complex structure, it may not be suitable for some programming languages. There is no Unicode version of the HOSTENT structure and members will return pointers to ANSI strings even if the application is compiled to use Unicode.

This function is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **DnsGetHostAddress** and **DnsGetHostName** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsGetHostByAddress, DnsGetHostInfo, DnsGetHostName, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord, DnsGetResolverAddress, DnsRegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostFile Function

```
INT WINAPI DnsGetHostFile(
   HCLIENT hClient,
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

The **DnsGetHostFile** function returns the name of the host file previously set using the **DnsSetHostFile** function. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **DnsGetHostAddress** and **DnsGetHostName** function.

Parameters

hClient

Handle to the client session.

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. A return value of zero indicates no host file has been specified or the function was unable to determine the file name. To get extended error information, call **DnsGetLastError**.

Remarks

This function returns the name of the host file that is cached in memory for the current thread. If an alternate host file has not been specified, this function will return the path to the default host file. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **DnsSetHostFile** function.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this function may be different than the default host file for the local system. To determine the file name for the default host file, use the **DnsGetDefaultHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetDefaultHostFile, DnsGetHostAddress, DnsGetHostName, DnsSetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostInfo Function

```
INT WINAPI DnsGetHostInfo(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **DnsGetHostInfo** function returns the HINFO record for the specified hostname. This information, if it is provided, typically specifies the operating system type and hardware platform.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the host information buffer is returned. A return value of zero indicates that no information is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

Many systems to do not maintain HINFO records for a site since that information can potentially be used to compromise system security. The information is typically used for administrative purposes with internal networks.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostByAddress, DnsGetHostByName, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord, DnsGetResolverAddress, DnsRegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostName Function

```
INT WINAPI DnsGetHostName(
   HCLIENT hClient,
   LPCTSTR lpszHostAddress,
   LPTSTR lpszHostName,
   INT nMaxLength
);
```

The **DnsGetHostName** function resolves the specified IP address, storing the fully qualified host name in the provided buffer.

Parameters

hClient

Handle to the client session.

lpszHostAddress

Pointer to a string that specifies an IPv4 or IPv6 formatted address.

lpszHostName

Pointer to the buffer that will contain the fully qualified domain name for the specified host. This buffer should be at least 64 characters in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the host name buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetHostName** function first looks to see if there is an entry in the local host file for the specified IP address, and if one exists, it will return the host name for that address. If you do not want to use the local host file at all, and only return an host name if a DNS query resolves the address, use the **DnsGetRecord** function and specify a record type of DNS_RECORD_PTR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsEnumHostAliases, DnsGetHostByAddress, DnsGetHostByName, DnsGetHostAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetHostServices Function

```
INT WINAPI DnsGetHostServices(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   INT nProtocol,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **DnsGetHostServices** function returns the WKS (Well Known Services) record for the specified hostname and protocol. This information, if it is provided, typically specifies the names of those services supported on the host.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nProtocol

The protocol for those services that information should be returned about. The following protocols are recognized:

Value	Description
INET_PROTOCOL_TCP	Services that use the Transmission Control Protocol (TCP)
INET_PROTOCOL_UDP	Services that use the User Datagram Protocol (UDP)

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the host services buffer is returned. A return value of zero indicates that no information is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

Many systems to do not maintain complete services records for a site since that information can potentially be used to compromise system security. An application should not depend on this information being available for any given record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostByAddress, DnsGetHostByName, DnsGetHostInfo, DnsGetMailExchange, DnsGetRecord, DnsGetResolverAddress, DnsRegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetLastError Function

```
DWORD WINAPI DnsGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last-error code value. Functions set this value by calling the **DnsSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **DnsGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **DnsSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those functions which call **DnsSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetErrorString, DnsSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetLocalAddress Function

```
INT WINAPI DnsGetLocalAddress(
   HCLIENT hClient,
   INT nAddressFamily,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **DnsGetLocalAddress** function returns the IP address for the local host.

hClient

Handle to the client session.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of DNS_ADDRESS_IPV4 specifies that the IPv4 address for the host should be returned. A value of DNS_ADDRESS_IPV6 specifies that the IPv6 address for the host should be returned. A value of DNS_ADDRESS_ANY specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host.

lpszAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 40 characters in length to accommodate both IPv4 and IPv6 addresses.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the host address buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetLocalAddress** function may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the ***nAddressFamily*** parameter as DNS_ADDRESS_IPV4 to prevent the possibility of an IPv6 address being returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostName, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetLocalDomain Function

```
INT WINAPI DnsGetLocalDomain(
   HCLIENT hClient,
   LPTSTR lpszDomain,
   INT nMaxLength
);
```

The **DnsGetLocalDomain** function copies the local domain name into the specified buffer. The value returned is the same value that was set with the **DnsSetLocalDomain** function. If no local domain name has been set, an empty string is returned.

Parameters

hClient

Handle to the client session.

lpszDomain

Pointer to the buffer that is used to store the local domain name. If no local domain name has been set, this buffer will be set to a zero-length string.

nMaxLength

The maximum number of bytes to copy into the buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the length of the domain name string. A return value of zero indicates that no local domain name has been set. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsRegisterServer, DnsSetLocalDomain

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetLocalName Function

```
INT WINAPI DnsGetLocalName(
   HCLIENT hClient,
   LPTSTR lpszLocalName,
   INT nMaxLength
);
```

The **DnsGetLocalName** function returns the local host name.

Parameters

hClient

Handle to the client session.

lpszLocalName

Pointer to a string buffer that will contain the local host name. It is recommended that this buffer be at least 64 characters in size.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is the length of the local host name. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostByAddress, DnsGetHostByName, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetMailExchange Function

```
INT WINAPI DnsGetMailExchange(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   LPINT lpnPreference,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **DnsGetMailExchange** function returns the mail exchange (MX) record information for the specified domain. This information, if it is provided, identifies a server responsible for processing mail for the given domain.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpnPreference

Pointer to the integer which will contain the preference for the specified mail exchange host.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the buffer is returned. A return value of zero indicates that no information is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The mail exchange record is typically used by mail delivery agents to determine what system is responsible for accepting mail addressed to a given domain. This function will return the first MX record provided by the server. Note that some domains may have multiple mail servers. To enumerate all of the mail exchange records for a domain, use the **DnsEnumMailExchanges** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostByAddress, DnsGetHostByName, DnsGetHostInfo, DnsGetHostServices, DnsGetRecord, DnsGetResolverAddress, DnsRegisterServer, DnsEnumMailExchanges

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetRecord Function

```
INT WINAPI DnsGetRecord(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   INT nRecordType,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **DnsGetRecord** function returns the specified record information for the given hostname.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nRecordType

The record type for the information that should be returned. The following record types are recognized:

Value	Description
DNS_RECORD_NONE	No record type
DNS_RECORD_ADDRESS	IPv4 host address
DNS_RECORD_NS	Authoritative nameserver
DNS_RECORD_CNAME	Canonical host name (alias)
DNS_RECORD_SOA	Start of Authority
DNS_RECORD_WKS	Well known services
DNS_RECORD_PTR	Domain name
DNS_RECORD_HINFO	Host information
DNS_RECORD_MINFO	Mailbox information
DNS_RECORD_MX	Mail exchange host
DNS_RECORD_TXT	Text strings
DNS_RECORD_AAAA	IPv6 host address
DNS_RECORD_LOC	Location information
DNS_RECORD_UINFO	User information
DNS_RECORD_UID	User ID
DNS_RECORD_GID	Group ID

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the information buffer is returned. A return value of zero indicates that no information for that record is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetRecord** function can be used to resolve a host name into an IPv4 address using the record type DNS_RECORD_ADDRESS, or an IPv6 address using the record type DNS_RECORD_AAAA. It can also be used to perform a reverse lookup and resolve an IP address into a host name by using the record type DNS_RECORD_PTR.

To determine the host that serves as the primary or master DNS for a zone, the record name should be specified as the domain name (e.g.: microsoft.com) and the record type should be DNS_RECORD_SOA. The value returned will the fully qualified domain name for host.

Note that this function does not reference a local host file when resolving host names or addresses. If the record lookup fails, this function will return an error even if there's an entry for the host in the file that has been specified by a call to **DnsSetHostFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsGetHostName, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetResolverAddress, DnsRegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetResolverAddress Function

```
INT WINAPI DnsGetResolverAddress(
    HCLIENT hClient,
    LPTSTR lpszAddress,
    INT nMaxLength
);
```

The **DnsGetResolverAddress** returns the address of the nameserver that resolved the last query.

Parameters

hClient

Handle to the client session.

lpszAddress

A null-terminated string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the function succeeds, the return value is the length of the address, not including the terminating null character. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostByAddress, DnsGetHostByName, DnsGetHostInfo, DnsGetHostServices, DnsGetMailExchange, DnsGetRecord, DnsRegisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetResolverOptions Function

```
DWORD WINAPI DnsGetResolverOptions(
    HCLIENT hClient
);
```

The **DnsGetResolverOptions** function returns the options that have been set for the client session.

Parameters

hClient

Handle to the client session

Return Value

If the function succeeds, the return value is the resolver options set for the client session. If the client handle is invalid or no resolver options have been specified, the function will return zero.

Remarks

The **DnsGetResolverOptions** function can be used to determine which resolver options have been specified for the client session. For a list of the available options, refer to the documentation for the **DnsSetResolverOptions** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsCreateHandle, DnsSetResolverOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetRetryCount Function

```
INT WINAPI DnsGetRetryCount(
    HCLIENT hClient
);
```

The **DnsGetRetryCount** returns the retry count for the current client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the retry count. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetTimeout, DnsSetRetryCount, DnsSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetServerAddress Function

```
INT WINAPI DnsGetServerAddress(
   HCLIENT hClient,
   INT nServer,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **DnsGetServerAddress** function returns the address of the registered nameserver.

Parameters

hClient

Handle the client session.

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **DnsRegisterServer** function when the nameserver is registered.

lpszAddress

A null-terminated string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the function succeeds, the return value is the length of the address, not including the terminating null character. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsCreateHandle, DnsGetResolverAddress, DnsGetServerPort, DnsRegisterServer, DnsUnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetServerPort Function

```
INT WINAPI DnsGetServerPort(
   HCLIENT hClient,
   INT nServer
);
```

The **DnsGetServerPort** function returns the port number registered to the specified nameserver.

Parameters

hClient

Handle the client session.

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **DnsRegisterServer** function when the nameserver is registered.

Return Value

If the function succeeds, the return value is the specified port number. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetResolverAddress, DnsGetServerAddress, DnsRegisterServer, DnsUnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetServiceName Function

```
INT WINAPI DnsGetServiceName(
   HCLIENT hClient,
   INT nServicePort,
   INT nServiceType,
   LPTSTR lpszServiceName,
   INT nMaxLength
);
```

The **DnsGetServiceName** function returns the service name assigned to the specified port number.

Parameters

hClient

Handle to the client session.

nServicePort

An integer which specifies the port number. This value must be a valid port number in the range of 1 through 65535.

nServiceType

An integer which specifies the protocol associated with the service. This may be one of the following values:

Value	Description
DNS_SERVICE_ANY	Match the first service entry for the specified port number regardless of the protocol used.
DNS_SERVICE_TCP	Match a service entry for the specified port number which uses the TCP protocol.
DNS_SERVICE_UDP	Match a service entry for the specified port number which uses the UDP protocol.

lpszServiceName

A null-terminated string buffer which will contain the name of the service assigned to the port number when the function returns. This parameter cannot be NULL and the buffer must be large enough to store the complete service name, including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Remarks

Port numbers can be assigned human readable service names to make them easier to reference. This function will return the name assigned to the specified port number. If the port number does not have an assigned name, this function will return zero and the last error code will be set to ST_ERROR_INVALID_SERVICE_NAME.

Most applications should use DNS_SERVICE_ANY to return the service name assigned to a port number. Services typically use the same name and port number regardless of the protocol used to establish a connection with them. However, specifying a protocol can enable your application to determine if the service supports the protocol. For example, calling this function with port number 21 and DNS_SERVICE_UDP would result in an error because FTP only supports TCP connections.

Service names are assigned by the Internet Assigned Numbers Authority (IANA) and the database of recognized service names can be returned by calling the **DnsGetDefaultServicesFile** function. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetDefaultHostFile, DnsGetDefaultServicesFile, DnsGetHostFile, DnsGetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetServicePort Function

```
INT WINAPI DnsGetServicePort(
   HCLIENT hClient,
   LPCTSTR lpszServiceName,
   INT nServiceType
);
```

The **DnsGetServicePort** function returns the port number assigned to the specified service.

Parameters

hClient

Handle to the client session.

lpszServiceName

A null-terminated string which specifies the service name. Service names are not case sensitive.

nServiceType

An integer which specifies the protocol associated with the service. This may be one of the following values:

Value	Description
DNS_SERVICE_ANY	Match the first service entry for the specified port number regardless of the protocol used.
DNS_SERVICE_TCP	Match a service entry for the specified port number which uses the TCP protocol.
DNS_SERVICE_UDP	Match a service entry for the specified port number which uses the UDP protocol.

Return Value

If the function succeeds, the return value is the port number assigned to the service name. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

Port numbers can be assigned human readable service names to make them easier to reference. This function will return the port number assigned to the specified service name. If the service name does not exist, this function will return DNS_ERROR and the last error code will be set to ST_ERROR_INVALID_SERVICE_NAME.

Most applications should use DNS_SERVICE_ANY to return the port number assigned to a service. Services typically use the same name and port number regardless of the protocol used to establish a connection with them. However, specifying a protocol can enable your application to determine if the service supports the protocol. For example, calling this function with the service name `"ftp"` and DNS_SERVICE_UDP would result in an error because FTP only supports TCP connections.

Service names are assigned by the Internet Assigned Numbers Authority (IANA) and the database of recognized service names can be returned by calling the **DnsGetDefaultServicesFile** function. The default location for this file is in a protected area of the Windows operating system and can only be modified by a process with administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetDefaultHostFile, DnsGetDefaultServicesFile, DnsGetHostFile, DnsGetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsGetTimeout Function

```
INT WINAPI DnsGetTimeout(
   HCLIENT hClient
);
```

The **DnsGetTimeout** function returns the timeout value for the current client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetRetryCount, DnsSetRetryCount, DnsSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsHostNameToUnicode Function

```
INT WINAPI DnsHostNameToUnicode(
   LPCTSTR lpszHostName,
   LPTSTR lpszUnicodeName,
   INT nMaxLength
);
```

The **DnsHostNameToUnicode** function converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the ***lpszUnicodeName*** string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsHostNameToUnicode** function will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this function, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the Unicode version of this function is used, the value returned in ***lpszUnicodeName*** will be a Unicode string using UTF-16 encoding. If the ANSI version of this function, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this function performs checks to ensure that the ***lpszHostName*** parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the DnsGetHostAddress function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsNormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsInitialize Function

```
BOOL WINAPI DnsInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **DnsInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **DnsInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **DnsUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **DnsInitialize** or **DnsUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsCloseHandle, DnsCreateHandle, DnsUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsMatchHostName Function

```
BOOL WINAPI DnsMatchHostName(
   HCLIENT hClient,
   LPCTSTR lpszHostName,
   LPCTSTR lpszHostMask
   BOOL bResolve
);
```

The **DnsMatchHostName** function matches a host name against one more strings that may contain wildcards.

Parameters

hClient

Handle to the client session.

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsMatchHostName** function provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the function would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the function would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetAddress, DnsGetHostAddress, DnsGetHostName, DnsGetLocalAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsNormalizeHostName Function

```
INT WINAPI DnsNormalizeHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **DnsNormalizeHostName** function returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the ***lpszNormalized*** string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsNormalizeHostName** function will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this function, it will be converted to an ASCII compatible format for domain names.

If the Unicode version of this function is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended that you use the Unicode version.

Although this function performs checks to ensure that the ***lpszHostName*** parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the DnsGetHostAddress function.

It is recommended that you use this function if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this function prior to calling the other DNS API functions that accept a host name as a parameter. Those functions already normalize the host name and perform checks to ensure it is in the correct format.

If the ***lpszHostName*** parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if

the IP address has a corresponding host name, use the DnsGetHostName method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetHostAddress, DnsGetHostName, DnsHostNameToUnicode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsRegisterServer Function

```
INT WINAPI DnsRegisterServer(
   HCLIENT hClient,
   INT nServer,
   LPCTSTR lpszHostAddress,
   INT nPort
);
```

The **DnsRegisterServer** function registers a nameserver with the current client session. The nameserver is used to resolve queries issued by the client, such as returning the IP address for a given host name. At least one nameserver must be registered by the client before queries are issued.

Parameters

hClient

Handle to the client session.

nServer

The index into the client nameserver table. This index, starting at 0, is used to specify which slot in the client's nameserver table will be used to store the nameserver information.

lpszHostAddress

A null-terminated string that specifies the IP address of the nameserver to be registered. Note that hostnames cannot be specified.

nPort

The port number that the specified nameserver is accepting queries on. This value may be set to zero, in which case it will use the default port value.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsCreateHandle, DnsGetServerAddress, DnsGetServerPort, DnsUnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsReset Function

```
INT WINAPI DnsReset(
   HCLIENT hClient
);
```

The **DnsReset** function resets the current state of the client session. The timeout and retry counts are set to their default values, the local domain name is cleared and all registered servers are removed from the client nameserver table.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsRegisterServer, DnsSetLocalDomain, DnsSetRetryCount, DnsSetTimeout, DnsUnregisterServer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsSetHostFile Function

```
INT WINAPI DnsSetHostFile(
   HCLIENT hClient,
   LPCTSTR lpszFileName
);
```

The **DnsSetHostFile** function specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **DnsGetHostAddress** and **DnsGetHostName** functions. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

hClient

Handle to the client session.

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the function succeeds, the return value is the number of entries in the host file. A return value of DNS_ERROR indicates failure. To get extended error information, call **DnsGetLastError**.

Remarks

This function loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this function again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **DnsGetHostFile** function. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **DnsGetDefaultHostFile** function. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetDefaultHostFile, DnsGetHostAddress, DnsGetHostFile, DnsGetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsSetLastError Function

```
VOID WINAPI DnsSetLastError(
   DWORD dwErrorCode
);
```

The **DnsSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread. A value of zero clears the last error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those functions which call **DnsSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **DnsGetLastError** function. The use of **DnsGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetErrorString, DnsGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsSetLocalDomain Function

```
INT WINAPI DnsSetLocalDomain(
   HCLIENT hClient,
   LPCTSTR lpszDomain
);
```

Parameters

hClient

Handle to the client session.

lpszDomain

Pointer to the string which contains the local domain name. This is used as a default value when a query does not explicitly specify a domain name.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DnsGetLocalDomain, DnsGetRetryCount, DnsGetTimeout, DnsReset, DnsSetRetryCount, DnsSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsSetResolverOptions Function

```
DWORD WINAPI DnsSetResolverOptions(
    HCLIENT hClient,
    DWORD dwOptions
);
```

The **DnsSetResolverOptions** function changes the resolver options for the specified client session.

Parameters

hClient

Handle to the client session.

dwOptions

An unsigned integer that specifies one or more resolver options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
DNS_OPTION_NONE	No additional resolver options specified. This is the default value, and it is recommended that most applications do not specify additional options unless the implications of doing so are understood.
DNS_OPTION_PROMOTE	Promotes the server that successfully completed the last query to the first server that will be used to resolve subsequent queries. This option can improve performance in some cases where one or more of the registered servers are non-responsive. This option takes precedence over the DNS_OPTION_ROTATE option.
DNS_OPTION_ROTATE	Enables a round-robin selection of nameservers when performing queries. Normally each nameserver is queried in the same order. This option rotates the available nameservers so a different server is used with each query.
DNS_OPTION_AUTHONLY	Require the answer from the nameserver to be authoritative, not from the server's cache. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_PRIMARY	Queries are only accepted from the primary nameserver. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_NORECURSE	Disable the sending of recursive queries to the nameserver. Specifying this option will disable the bit in the DNS request header that specifies recursion is desired.
DNS_OPTION_NOSEARCH	Disable additional queries of higher domains in the search list if the host name cannot be resolved. If this option is specified, and the host name cannot be resolved using the local domain name an error is

	returned immediately. This option is ignored if no local domain has been specified or if the DNS_OPTION_NOSUFFIX option has been specified.
DNS_OPTION_NOSUFFIX	Disable additional queries using the local domain name if the host name is not a fully qualified domain name and cannot be resolved. This option is ignored if no local domain has been specified.
DNS_OPTION_NONAMECHECK	Disable checking the host name for invalid characters, such as the underscore and control characters. By default, host names are checked to ensure they're valid before submitting a query to the nameserver.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsSetResolverOptions** function changes the resolver options for the specified client session, modifying how nameserver queries are processed. It is recommended that most applications do not specify any resolver options and use the default behavior. Specifying these options without understanding how they can affect standard queries can result in unexpected failures. In particular, caution should be used when specifying the DNS_OPTION_NORECURSE and DNS_OPTION_NOSEARCH options as they change the normal process of resolving a host name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsCreateHandle, DnsGetResolverOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsSetRetryCount Function

```
INT WINAPI DnsSetRetryCount(
   HCLIENT hClient,
   INT nRetries
);
```

The **DnsSetRetryCount** function sets the number of attempts that the client will make attempting to resolve a query. When used in conjunction with the **DnsSetTimeout** function, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

hClient

Handle to the client session.

nRetries

The number of attempts the client will make, per nameserver, to resolve a query.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetRetryCount, DnsGetTimeout, DnsSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsSetTimeout Function

```
INT WINAPI DnsSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **DnsSetTimeout** function sets the number of seconds that the client will wait for a response from a nameserver. The timeout value is used each time a server in the client's nameserver table is queried. When used in conjunction with the **DnsSetRetryCount** function, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds until the client times out waiting for a response from a nameserver.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, with two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetRetryCount, DnsGetTimeout, DnsSetRetryCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsUninitialize Function

```
VOID WINAPI DnsUninitialize();
```

The **DnsUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **DnsInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **DnsUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **DnsUninitialize** for every successful call to **DnsInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsInitialize, DnsCloseHandle, DnsCreateHandle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DnsUnregisterServer Function

```
INT WINAPI DnsUnregisterServer(
    HCLIENT hClient,
    INT nServer
);
```

The **DnsUnregisterServer** function removes the specified nameserver information from the client. Unregistering a server prevents the client from using that server to satisfy subsequent DNS queries.

Parameters

hClient

Handle to the client session.

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **DnsRegisterServer** function when the nameserver is registered.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csdnsv12.lib

See Also

DnsGetServerAddress, DnsGetServerPort, DnsRegisterServer, DnsReset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Data Structures

- HOSTENT
- INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HOSTENT

The **HOSTENT** structure contains information about a domain name, including its IP address and any aliases which may be defined.

```
typedef struct _HOSTENT
{
   char *  h_name;
   char ** h_aliases;
   short   h_addrtype;
   short   h_length;
   char ** h_addr_list;
} HOSTENT, *LPHOSTENT;
```

Members

h_name

The fully qualified domain name (FQDN) that caused the nameserver server to return a reply.

h_aliases

A NULL-terminated array of alternate names.

h_addrtype

The type of address being returned.

h_length

The length, in bytes, of each address.

h_addr_list

A NULL-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h_addr** is defined to be **h_addr_list**[0] for compatibility with older software.

Remarks

This structure is intended to be used for backwards compatibility with legacy applications and should not be used with new applications. You cannot use this structure to obtain information for a host which is assigned an IPv6 address. Applications should use the **DnsGetHostAddress** and **DnsGetHostName** functions which support both IPv4 and IPv6 addresses.

The application must never attempt to modify this structure or free any of its members. There is no Unicode version of this structure, string pointers will always specify ANSI strings, even if the application is compiled to use Unicode. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before calling any other functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: winsock2.h.

See Also

DnsGetHostAddress, DnsGetHostByAddress, DnsGetHostByName, DnsGetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
     INT    ipFamily;
     BYTE   ipNumber[16];
}  INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Domain Name Service Constants

Value	Description
DNS_RECORD_NONE	No record type
DNS_RECORD_ADDRESS	Host address
DNS_RECORD_NS	Authoritative nameserver
DNS_RECORD_CNAME	Canonical name (alias)
DNS_RECORD_SOA	Start of Authority
DNS_RECORD_WKS	Well known services
DNS_RECORD_PTR	Domain name
DNS_RECORD_HINFO	Host information
DNS_RECORD_MINFO	Mailbox information
DNS_RECORD_MX	Mail exchange host
DNS_RECORD_TXT	Text strings
DNS_RECORD_LOC	Location information
DNS_RECORD_UINFO	User information
DNS_RECORD_UID	User ID
DNS_RECORD_GID	Group ID

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Encoding Library

A utility library for encoding, compressing, and encrypting files.

Reference

- Functions

Library Information

File Name	CSNCDV12.DLL
Version	12.0.3024.1286
LibID	B1B7354C-B0D2-4BD1-858B-9B0423768198
Import Library	CSNCDV12.LIB
Dependencies	None
Standards	RFC 1738, RFC 1951, RFC 2045, RFC 8089

Overview

The File Encoding library is a versatile collection of functions for transforming and processing file and memory data. Originally focused on encoding binary files into text formats such as Base64 and Uuencode for use in email and newsgroups, the library has since expanded into a broader utility toolkit. It now includes support for a variety of encoding formats including Base32, Base58, and Base85, along with functions for decoding, compression, and expansion of files and buffers.

In addition to encoding and decoding, the library provides robust support for AES-256 encryption and decryption of strings, files, and memory blocks. It also includes utilities for comparing file contents, computing message digests, converting text files between different Unicode and legacy code pages, and determining the encoding of existing files. For applications that require unique identifiers, the library offers functions to create, parse, and manipulate UUIDs (GUIDs), including sequential, namespace-based, and custom values. There are also helpers for working with file URLs, verifying Unicode content, and generating secure tokens.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on

ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Encoding Functions

Function	Description
AesDecryptBuffer	Decrypt a buffer previously encrypted using the AesEncryptBuffer function
AesDecryptFile	Decrypt a file previously encrypted using the AesEncryptFile function
AesDecryptString	Decrypt a string previously encrypted using the AesEncryptString function
AesEncryptBuffer	Encrypt a memory buffer using 256-bit AES encryption
AesEncryptFile	Encrypt the contents of a file using 256-bit AES encryption
AesEncryptString	Encrypt a null-terminated string using 256-bit AES encryption
CompareFile	Compare the contents of two files to determine if they are identical
CompareFileEx	Compare the contents of two files or the contents of a file against message digest
CompressBuffer	Compress the contents of the specified buffer
CompressFile	Compress the contents of the specified file
CompressFileEx	Compress the contents of the specified file with additional options
ConvertTextFile	Convert the contents of a text file from one code page to another
CreateCustomGuid	Create a unique GUID which includes a 32-bit user-defined value
CreateGuidFromString	Parse a string which contains a GUID and return its value
CreateHexStringFromGuid	Create a string of hexadecimal digits from the specified GUID
CreateNamespaceGuid	Create a deterministic namespace identifier from the specified GUID
CreateSequentialGuid	Create a unique sequential GUID which can be sorted lexicographically
CreateStringFromGuid	Create a string representation of the specified GUID
CreateUniqueToken	Create a unique cryptographically secure token string
DecodeBuffer	Decode an encoded string, storing the result in the specified buffer
DecodeFile	Decode the contents of a file using the specified decoding method
EncodeBuffer	Encode the contents of a buffer using the specified encoding method
EncodeFile	Encode the contents of a file using the specified encoding method
ExpandBuffer	Expand the contents of a previously compressed buffer
ExpandFile	Expand the contents of a previously compressed file
ExpandFileEx	Expand the contents of a previously compressed file with additional options
FilterHtmlFile	Filters the contents of an HTML file and returns the text in a string buffer
FilterHtmlText	Removes HTML formatting from a string and returns simplified plain text
FilterMarkdownText	Removes Markdown formatting from a string and returns simplified plain text
FilterUnicodeText	Filters and replaces confusable characters and symbols in a Unicode string
GetCustomGuidValue	Return the custom 32-bit value used to create a Version 8 GUID.
GetFileCodePage	Determine what code page was used to create the specified text file

GetFileNameUrl	Return a file URL which conforms to the standard file URI scheme
GetFileTimeFromGuid	Return the timestamp from a sequential GUID in a FILETIME structure
GetLocalMachineGuid	Return a unique identifier for the local machine
GetMessageDigest	Compute a message digest for a buffer, string or data file
GetSystemTimeFromGuid	Return the timestamp from a sequential GUID in a SYSTEMTIME structure
GetTimeFromGuid	Returns the 64-bit timestamp value from a sequential GUID
GetUrlFileName	Parse a file URL and return the corresponding local file name or UNC path
IsEqualGuidString	Compare two strings containing GUIDs and determine if they are the same value
IsHtmlText	Determine whether a string appears to contain HTML formatting
IsMarkdownText	Determine whether a string appears to contain Markdown formatting
IsUnicodeConfusable	Determine if a Unicode string contains confusable characters
IsUnicodeText	Determine if a text buffer contains valid Unicode characters
IsValidFileUrl	Determine if a file URL is valid and optionally check if the file exists
NormalizeUrl	Validate and normalize a string which specifies a web address or local file URL
UnicodeDecodeText	Decode UTF-8 encoded text and return a localized string
UnicodeEncodeText	Encode a text buffer and return a UTF-8 encoded string

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AesDecryptBuffer Function

```
BOOL WINAPI AesDecryptBuffer(
   LPCTSTR lpszPassword,
   LPCVOID lpvInputBuffer,
   DWORD dwInputSize
   LPVOID lpvOutputBuffer,
   LPDWORD lpdwOutputSize
);
```

The **AesDecryptBuffer** function decrypts the contents of a memory buffer.

Parameters

lpszPassword

A null-terminated string of characters that will be used to generate the decryption key. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used to decrypt the data. Password strings that exceed 215 characters will be truncated.

lpvInputBuffer

The buffer which contains the data to be decrypted.

dwInputSize

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

The buffer which will contain the decrypted data when the function returns. This memory must be allocated by the application and be large enough to contain all of the decrypted data. If the output buffer is not large enough, the function will fail. This parameter cannot be NULL.

lpdwOutputSize

The number of bytes that may be copied into the output buffer. This parameter must be initialized to a non-zero value. When the function returns, the actual number of bytes of decrypted data is returned in this parameter. This parameter cannot be NULL.

Return Value

A non-zero value is returned if the data was successfully decrypted. A zero value indicates that the data could not be decrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesDecryptBuffer** function will decrypt a block of memory using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the decrypted data to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process. The value of the ***lpszPassword*** parameter must be identical to the value used to encrypt the data using the **AesEncryptBuffer** function.

This function cannot be used to decrypt data that was encrypted using another third-party library or utility. It can only be used to decrypt data that was previously encrypted using the **AesEncryptBuffer** function.

If you wish to decrypt the contents of a file, use the **AesDecryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other

countries. If the required cryptographic provider is not available, the function will fail.

Example

```
#define MAX_BUFFER_SIZE 1024

BYTE lpInputBuffer[MAX_BUFFER_SIZE] = "The quick brown fox jumps over the lazy
dog.";
BYTE lpEncryptedBuffer[MAX_BUFFER_SIZE + 256];
BYTE lpDecryptedBuffer[MAX_BUFFER_SIZE];

DWORD dwInputSize = (DWORD)strlen((LPCSTR)lpInputBuffer);
DWORD dwEncryptedSize = sizeof(lpEncryptedBuffer);
DWORD dwDecryptedSize = sizeof(lpDecryptedBuffer);

LPCTSTR lpszPassword = _T("MySecretPassword");

BOOL bResult = FALSE;

// Encrypt the data
bResult = AesEncryptBuffer(lpszPassword,
                           lpInputBuffer,
                           dwInputSize,
                           lpEncryptedBuffer,
                           &dwEncryptedSize);

if (!bResult)
{
    _tprintf(_T("Encryption failed, error 0x%08lX\n"), GetLastError());
    return;
}

// Decrypt the data
bResult = AesDecryptBuffer(lpszPassword,
                           lpEncryptedBuffer,
                           dwEncryptedSize,
                           lpDecryptedBuffer,
                           &dwDecryptedSize);

if (!bResult)
{
    _tprintf(_T("Decryption failed, error 0x%08lX\n"), GetLastError());
    return;
}

// Verify the decrypted data matches the original input
// Note that encrypted data is binary and may contain embedded nulls
if (dwDecryptedSize == dwInputSize &&
    memcmp(lpInputBuffer, lpDecryptedBuffer, dwInputSize) == 0)
{
    _tprintf(_T("Success: Decrypted data matches original input.\n"));
}
else
{
    _tprintf(_T("Error: Decrypted data does not match original input.\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AesDecryptFile, AesDecryptString, AesEncryptBuffer, AesEncryptFile, AesEncryptString, CompressBuffer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AesDecryptFile Function

```
BOOL WINAPI AesDecryptFile(
   LPCTSTR lpszPassword,
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile
);
```

The **AesDecryptFile** function decrypts the contents of a file.

Parameters

lpszPassword

A null-terminated string of characters that will be used to generate the decryption key. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used to decrypt the data. Password strings that exceed 215 characters will be truncated.

lpszInputFile

A null-terminated string which specifies the name of the file to be decrypted. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

A null-terminated string which specifies the name of the file that will contain the decrypted data. If the file exists, it will be overwritten. It must be a regular file that can be opened for writing by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

Return Value

A non-zero value is returned if the string was successfully encrypted. A zero value indicates that the string could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesDecryptFile** function will decrypt the contents of a file using a 256-bit AES (Advanced Encryption Standard) algorithm and stores the decrypted data in the specified output file. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process. The ***lpszPassword*** value must be identical to the value used to encrypt the file using the **AesEncryptFile** function.

This function cannot be used to decrypt a file that was encrypted using another third-party library or utility. It can only be used to decrypt a file that was previously encrypted using the **AesEncryptFile** function.

A temporary file is created during the decryption process and the output file is created or overwritten only if the input file could be successfully decrypted. If the decryption fails, no output file will be created.

If you wish to decrypt the contents of a memory buffer or string, use the **AesDecryptBuffer** or **AesDecryptString** functions.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AesDecryptBuffer, AesDecryptString, AesEncryptFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AesDecryptString Function

```
LONG WINAPI AesDecryptString(
   LPCTSTR lpszPassword,
   LPCTSTR lpszInputString,
   LPTSTR lpszOutputString,
   LONG nMaxLength
);
```

The **AesDecryptString** function decrypts the contents of a string.

Parameters

lpszPassword

A null-terminated string of characters that will be used to generate the decryption key. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used to decrypt the data. Password strings that exceed 215 characters will be truncated.

lpszInputString

A null-terminated string which contains the data to be decrypted. The encrypted input data must be base64 encoded and identical to the encrypted string returned by the **AesEncryptString** function. If this parameter is NULL or points to an empty string the function will fail.

lpszOutputString

The buffer which will contain the decrypted string data. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied to the output string buffer. The output buffer must be large enough to store the complete decrypted string and is terminated with a null character. This value must be greater than zero. If the output string buffer is not large enough, the function will fail.

Return Value

The number of characters of decrypted text copied to the string buffer, not including the terminating null character. A zero value indicates that the string could not be decrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesDecryptString** function will decrypt a string using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the decrypted string o the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process. The ***lpszPassword*** value must be identical to the value used to encrypt the data using the **AesEncryptString** function.

Due to how the SHA-256 hash is generated, this function cannot be used to decrypt strings that were encrypted using another third-party library. It can only be used to decrypt strings that were previously encrypted using **AesEncryptString**.

If you wish to decrypt the contents of a file, use the **AesDecryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Example

```
LPCTSTR lpszPassword = _T("NFr-E{Ki3_1w0iV+LI@z");
TCHAR szDecryptedText[MAX_STRING_LENGTH];

INT nLength = AesDecryptString(lpszPassword,
                               szEncryptedText,
                               szDecryptedText,
                               MAX_STRING_LENGTH);

if (nLength > 0)
{
    _tprintf(_T("The decrypted string is \"%s\"\n"), szDecryptedText);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AesDecryptBuffer, AesDecryptFile, AesEncryptString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AesEncryptBuffer Function

```
BOOL WINAPI AesEncryptBuffer(
   LPCTSTR lpszPassword,
   LPCVOID lpvInputBuffer,
   DWORD dwInputSize
   LPVOID lpvOutputBuffer,
   LPDWORD lpdwOutputSize
);
```

The **AesEncryptBuffer** function encrypts the contents of a memory buffer.

Parameters

lpszPassword

A null-terminated string of characters that will be used to generate the encryption key. Passwords are case sensitive and must match exactly when decrypting data that was previously encrypted using this function. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used. Password strings that exceed 215 characters will be truncated.

lpvInputBuffer

The buffer which contains the data to be encrypted.

dwInputSize

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

The buffer which will contain the encrypted data when the function returns. This memory must be allocated by the application and be large enough to contain all of the encrypted data. The amount encrypted data returned will always be larger than the amount of data specified by ***dwInputSize***. If the output buffer is not large enough, the function will fail. This parameter cannot be NULL.

lpdwOutputSize

The number of bytes that may be copied into the output buffer. This parameter must be initialized to a non-zero value. When the function returns, the actual number of bytes of encrypted data is returned in this parameter. This parameter cannot be NULL.

Return Value

A non-zero value is returned if the data was successfully encrypted. A zero value indicates that the data could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesEncryptBuffer** function will encrypt a block of memory using a 256-bit AES (Advanced Encryption Standard) algorithm and returns the encrypted data to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. A library-defined internal value is incorporated when deriving the encryption key to inhibit casual decryption using external tools. This function is intended as a convenience for encrypting application data using a password-derived key. It is not designed as a general-purpose encrypted data format for interoperability with other libraries or utilities. Applications should use the corresponding **AesDecryptBuffer** function to decrypt the data.

It is recommended that most applications specify a password value. If the ***lpszPassword*** parameter is NULL or specifies a zero-length string, any application using this function with a NULL or empty password will be able to decrypt the data using the **AesDecryptBuffer** function without specifying a

password. This behavior is intended for convenience and should not be considered secure for protecting sensitive data. If the Unicode version of this function is called, the *lpszPassword* value will be encoded using UTF-8 prior to the hash value being generated.

The amount of encrypted data returned by this function will always be somewhat larger than the original unencrypted data. If your application dynamically allocates a buffer to store the encrypted data, it is recommended that you allocate a buffer that is at least 256 bytes larger than the input data. Allocating a significantly larger buffer is typically unnecessary. If the output buffer provided is not large enough, the function will fail and the **GetLastError** function will return ERROR_INSUFFICIENT_BUFFER.

The encrypted data returned by this function can contain embedded nulls and should be typically handled as byte values. If you wish to encrypt strings and store the encrypted values (for example, encrypting an API key and storing it in a configuration file used by your application), use the **AesEncryptString** function. It will perform the same 256-bit AES encryption, but return the encrypted data as a base64 encoded string rather than binary data.

If your application is also using the **CompressBuffer** function to compress the data, it is recommended that you call **CompressBuffer** before calling **AesEncryptBuffer**. You will typically achieve a better compression rate on unencrypted data than attempting to compress data which has been encrypted with this function.

If you wish to encrypt the contents of a file, use the **AesEncryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Example

```
#define MAX_BUFFER_SIZE 1024

BYTE lpInputBuffer[MAX_BUFFER_SIZE] = "The quick brown fox jumps over the lazy
dog.";
BYTE lpEncryptedBuffer[MAX_BUFFER_SIZE + 256];
BYTE lpDecryptedBuffer[MAX_BUFFER_SIZE];

DWORD dwInputSize = (DWORD)strlen((LPCSTR)lpInputBuffer);
DWORD dwEncryptedSize = sizeof(lpEncryptedBuffer);
DWORD dwDecryptedSize = sizeof(lpDecryptedBuffer);

LPCTSTR lpszPassword = _T("MySecretPassword");

BOOL bResult = FALSE;

// Encrypt the data
bResult = AesEncryptBuffer(lpszPassword,
                           lpInputBuffer,
                           dwInputSize,
                           lpEncryptedBuffer,
                           &dwEncryptedSize);

if (!bResult)
{
    _tprintf(_T("Encryption failed, error 0x%08lX\n"), GetLastError());
    return;
}
```

```
// Decrypt the data
bResult = AesDecryptBuffer(lpszPassword,
                           lpEncryptedBuffer,
                           dwEncryptedSize,
                           lpDecryptedBuffer,
                           &dwDecryptedSize);

if (!bResult)
{
    _tprintf(_T("Decryption failed, error 0x%08lX\n"), GetLastError());
    return;
}

// Verify the decrypted data matches the original input
// Note that encrypted data is binary and may contain embedded nulls
if (dwDecryptedSize == dwInputSize &&
    memcmp(lpInputBuffer, lpDecryptedBuffer, dwInputSize) == 0)
{
    _tprintf(_T("Success: Decrypted data matches original input.\n"));
}
else
{
    _tprintf(_T("Error: Decrypted data does not match original input.\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AesDecryptBuffer, AesDecryptFile, AesDecryptString AesEncryptFile, AesEncryptString, CompressBuffer

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AesEncryptFile Function

```
BOOL WINAPI AesEncryptFile(
   LPCTSTR lpszPassword,
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile
);
```

The **AesEncryptFile** function encrypts the contents of a file.

Parameters

lpszPassword

A null-terminated string of characters that will be used to generate the encryption key. Passwords are case sensitive and must match exactly when decrypting data that was previously encrypted using this function. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used. Password strings that exceed 215 characters will be truncated.

lpszInputFile

A null-terminated string which specifies the name of the file to be encrypted. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

A null-terminated string which specifies the name of the file that will contain the encrypted data. If the file exists, it will be overwritten. It must be a regular file that can be opened for writing by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

Return Value

A non-zero value is returned if the string was successfully encrypted. A zero value indicates that the string could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesEncryptFile** function will encrypt the contents of a file using a 256-bit AES (Advanced Encryption Standard) algorithm and returns the encrypted data to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. A library-defined internal value is incorporated when deriving the encryption key to inhibit casual decryption using external tools. This function is intended as a convenience for encrypting files using a password-derived key. It is not designed as a general-purpose function for interoperability with other libraries or platforms. Applications should use the corresponding **AesDecryptFile** function to decrypt the file.

It is recommended that most applications specify a password value. If the *lpszPassword* parameter is NULL or specifies a zero-length string, any application using this function with a NULL or empty password will be able to decrypt the data using the **AesDecryptFile** function without specifying a password. This behavior is intended for convenience and should not be considered secure for protecting sensitive data. If the Unicode version of this function is called, the *lpszPassword* value will be encoded using UTF-8 prior to the hash value being generated.

A temporary file is created during the encryption process and the output file is created or overwritten only if the input file could be successfully encrypted. If the encryption fails, no output file will be created.

The input file contents will always be processed as a binary data stream. If you use this function to encrypt a text file, the output file will contain binary characters, not printable text. If you wish to transfer or store the encrypted data as text, it should be encoded using the **EncodeFile** function after calling **AesEncryptFile**.

If your application is also using the **CompressFile** function to compress the data, it is recommended that you call **CompressFile** before calling **AesEncryptFile**. You will typically achieve a better compression rate on unencrypted data than than attempting to compress data which has been encrypted with this function.

If you wish to encrypt the contents of a memory buffer or string, use the **AesEncryptBuffer** or **AesEncryptString** functions.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AesDecryptFile, AesEncryptBuffer, AesEncryptString, CompressFile, EncodeFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

AesEncryptString Function

```
LONG WINAPI AesEncryptString(
   LPCTSTR lpszPassword,
   LPCTSTR lpszInputString,
   LONG nInputLength
   LPTSTR lpszOutputString,
   LONG nMaxLength
);
```

The **AesEncryptString** function encrypts the contents of a string.

Parameters

lpszPassword

A null-terminated string of characters that will be used to generate the encryption key. Passwords are case sensitive and must match exactly when decrypting data that was previously encrypted using this function. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used. Password strings that exceed 215 characters will be truncated.

lpszInputString

A null-terminated string which contains the text to be encrypted. If the Unicode version of this function is called, the string is automatically UTF-8 encoded prior to being encrypted.

nInputLength

The number of characters in the input string which will be encrypted. If this value is -1, the length of the input string is determined automatically by counting the number of of characters up to the terminating null byte. Note that with Unicode strings, this value must represent the number of characters in the string, not the number of bytes allocated for the string.

lpszOutputString

The buffer which will contain a base64 encoded encrypted string when the function returns. This memory must be allocated by the application and be large enough to contain all the encrypted data. The amount encrypted data returned will always be larger than the amount of data specified by ***nInputLength***. If the output string buffer is not large enough, the function will fail. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied to the output string buffer. The output buffer must be large enough to store the complete encrypted string which is encoded using base64 and terminated with a null character. This value must be greater than zero. If the output string buffer is not large enough, the function will fail.

Return Value

The number of characters of encrypted text copied to the string buffer, not including the terminating null character. A zero value indicates that the string could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesEncryptString** function will encrypt a string using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the encrypted data as a base64 encoded string to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. The identical password is required to decrypt the data

using the **AesDecryptString** function.

It is recommended that most applications specify a password value. If the *lpszPassword* parameter is NULL or specifies a zero-length string, a default internal hash value is used. This means that any other application which uses a NULL password value will be able to decrypt the data. If the Unicode version of this function is called, the *lpszPassword* value will be encoded using UTF-8 prior to the hash value being generated.

Due to how the SHA-256 hash is generated, the encrypted data cannot be decrypted using another third-party library with the same password value. It can only be decrypted using the **AesDecryptString** function.

The amount of encrypted data returned by this function will always be larger than original unencrypted data. If your application dynamically allocates a block of memory to store the encrypted string, provide a maximum output string size that is at least several hundred bytes larger than the unencrypted data. If the output string is not large enough, the function will fail and the **GetLastError** function will return ERROR_INSUFFICIENT_BUFFER.

The string provided to this function cannot contain embedded nulls and and should not be used to encrypt binary data. If you wish to encrypt binary data, use the **AesEncryptBuffer** function. It will perform the same 256-bit AES encryption and return the encrypted data into an buffer provided by the caller.

If you wish to encrypt the contents of a file, use the **AesEncryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Example

```
LPCTSTR lpszPassword = _T("NFr-E{Ki3_1w0iV+LI@z");
LPCTSTR lpszPlainText = _T("The quick brown fox jumped over the lazy dog.");
TCHAR szEncryptedText[MAX_STRING_LENGTH];

INT nLength = AesEncryptString(lpszPassword,
                               lpszPlainText,
                               -1,
                               szEncryptedText,
                               MAX_STRING_LENGTH);

if (nLength > 0)
{
    _tprintf(_T("The encrypted string is \"%s\"\n"), szEncryptedText);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

AesDecryptString, AesEncryptBuffer, AesEncryptFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CompareFile Function

```
BOOL WINAPI CompareFile(
   LPCTSTR lpszFile1,
   LPCTSTR lpszFile2
);
```

The **CompareFile** function compares the contents of two files to determine if they are identical.

Parameters

lpszFile1

A null-terminated string which specifies the name of the first file to compare. This parameter must specify the name of a file which exists and can be opened for read access by the current process. If the file name specifies a directory or device name, the function will fail. This parameter cannot be NULL or a zero-length string. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters after the variable expansion, they will be removed.

lpszFile2

A null-terminated string which specifies the name of the second file to compare. This parameter must specify the name of a file which exists and can be opened for read access by the current process. If the file name specifies a directory or device name, the function will fail. This parameter cannot be NULL or a zero-length string. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters after the variable expansion, they will be removed.

Return Value

A non-zero value is returned if the two files are identical. A value of zero indicates the files are different or an error has occurred. Check the value returned by the **GetLastError** function to determine the reason the file comparison was not successful. If the files are identical, this function will clear the last error code for the current thread.

Remarks

The **CompareFile** function can be used compare the contents of two files to determine if they are the same. If you need to perform a more complex file comparison, such as comparing the file creation or modification timestamps or verifying the file contents match a computed hash value, use the **CompareFileEx** function.

If the *lpszFile1* and *lpszFile2* file names resolve to the same file on the disk, this function will succeed without checking the contents of the file. This check is performed immediately after the file names have been expanded and converted to absolute paths. The function will still return an error if the file not does actually exist or is not a regular file.

To avoid potential problems with relative pathing in multithreaded applications, it is recommended you use absolute paths whenever possible. If you do not specify an absolute path, the file locations will be considered relative to the current working directory for the process. Because this is shared between all threads, it is possible for one thread to change the current working directory at any time and this can create inconsistent results. If you must use relative paths in a multithreaded application, use a mutex or critical section to ensure other threads cannot change the current working directory for the process while performing the file comparison.

This function will normalize the file names provided by the caller and perform checks to make sure the

names do not contain illegal characters. If the name includes quotes, wildcard characters or other invalid symbols the function will fail and return zero with the last error code set to ST_ERROR_INVALID_FILE_NAME. It is permitted to use forward slashes in path names and these will be converted to the standard backslash character used with Windows paths. This function cannot be used to compare the contents of NTFS alternate data streams.

This function is defined using an alias which automatically selects an ANSI or Unicode version based on the value of the UNICODE preprocessor macro. If you use the ANSI version of this function, the expanded file names will always be considered UTF-8 encoded strings, regardless of the current locale of the application. If your application uses file names which contain non-ASCII characters, you should always use the Unicode version of the function, even if your project is configured to use a multi-byte character set.

Example

```
// Compare the contents of two files
LPCTSTR lpszFile1 = _T("%USERPROFILE%\\AppName\\ProjectData.db");
LPCTSTR lpszFile2 = _T("%TEMP%\\BackupData.db");

BOOL bIdentical = CompareFile(lpszFile1, lpszFile2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareFileEx, CompressFile, GetMessageDigest, ExpandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CompareFileEx Function

```
BOOL WINAPI CompareFileEx(
   LPCTSTR lpszFile1,
   LPCTSTR lpszFile2,
   DWORD dwOptions,
   LPCTSTR lpszDigest
);
```

The **CompareFileEx** function compares the contents of two files with an optional message digest.

Parameters

lpszFile1

A null-terminated string which specifies the name of the first file to compare. This parameter must specify the name of a file which exists and can be opened for read access by the current process. If the file name specifies a directory or device name, the function will fail. This parameter cannot be NULL or a zero-length string. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters after the variable expansion, they will be removed.

lpszFile2

A null-terminated string which specifies the name of the second file to compare. If this parameter is NULL or a zero-length string, the ***lpszDigest*** parameter must specify a valid message digest and the contents of the first file will be compared against that digest. If this parameter specifies a file name, the file must exist and the current process must be able to open it for read access. If the file name specifies a directory or device name, the function will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters after the variable expansion, they will be removed.

dwOptions

An unsigned integer value which specifies one or more option flags which will control how the files are compared. It may be one or more of the following values:

Value	Description
COMPARE_FILE_DEFAULT	Use the default comparison option, which is to perform a byte comparison of both files. If the ***lpszDigest*** parameter is not NULL or a zero-length string, the function will compare the message digest against the contents of the file. If the ***lpszDigest*** parameter has a value and no specific hash algorithm has been specified as one of the options, the hash type will be automatically determined based on its value
COMPARE_FILE_CONTENTS	Perform a byte comparison of the two files. This option requires both files be specified. The current process must be able to open both files for exclusive read access. If either file is opened for write access, the function will fail. The ***lpszFile2*** parameter must specify a valid file name if this option is used.
COMPARE_FILE_TIME_CREATED	Compare the file creation times for both files and only

	consider them to be identical if the creation times are identical. If this option is specified and the creation times are different, the function will return zero even if the contents of the two files are identical. The ***lpszFile2*** parameter must specify a valid file name if this option is used.
COMPARE_FILE_TIME_MODIFIED	Compare the file modification times for both files and only consider them to be identical if the modification times are identical. If this option is specified and the modification times are different, the function will return zero even if the contents of the two files are identical. The ***lpszFile2*** parameter must specify a valid file name if this option is used.
COMPARE_FILE_DIGEST_MD5	Use the MD5 algorithm to compute 128-bit hash value for the contents of the first file and compare this against the specified digest string. This algorithm is the most efficient method to compute a hash to verify data integrity, however it is not considered cryptographically secure and hash collision is possible. When this option is used the ***lpszDigest*** parameter must specify a valid MD5 hash as a 32 digit hexadecimal string.
COMPARE_FILE_DIGEST_SHA1	Use the SHA-1 algorithm to compute 160-bit hash value for the contents of the first file and compare this against the specified digest string. This algorithm is comparable to the MD5 algorithm and has a lower chance of hash collision; however, it is not considered cryptographically secure and hash collision is possible. When this option is used the ***lpszDigest*** parameter must specify a valid SHA-1 hash as a 40 digit hexadecimal string.
COMPARE_FILE_DIGEST_SHA256	Use the SHA-256 algorithm to compute a 256-bit hash value for the contents of the first file and compare this against the specified digest string. This is a cryptographically secure algorithm which is more compute intensive than either MD5 or SHA-1, but the chance of a hash collision is extremely low and typically not a concern for most applications. When this option is used the ***lpszDigest*** parameter must specify a valid SHA-256 hash as a 64 digit hexadecimal string.

lpszDigest

A null-terminated string which specifies a sequence of hexadecimal numbers which are used to compare against the computed hash value for the file. If this parameter is not NULL and no option is specified to indicate the hash type, it will be inferred from the length of the string. If this parameter is NULL or a zero-length string, it will be ignored. The case of the hexadecimal digits in the string are ignored and it is permitted, but not required, to prefix the string with either "0x" or "&H" to indicate it is a hexadecimal value. If the string does not specify a valid hexadecimal number or is not a value computed using one of the supported hash algorithms, the function will fail.

Return Value

A non-zero value is returned if the files are identical using the specified criteria. A value of zero indicates the files are different or an error has occurred. Check the value returned by the **GetLastError** function to determine the reason the file comparison was not successful. If the files are identical, this function will clear the last error code for the current thread.

Remarks

The **CompareFileEx** function can be used to either compare the contents of two files or compare the contents of a file against a pre-computed hash value to determine if they match. In addition to comparing file contents, the function can also check the file creation and/or modification times to ensure they are the same. You can use the **GetMessageDigest** function to compute a hash value for a file.

In most cases, an application will specify either two files to compare or a message digest value. If you call the function using two file names and a message digest, the contents of both files must be identical and the computed hash value must match the specified message digest. If any of the requirements are are not satisfied, the function will return zero (false). For example, if the actual contents of the two files are identical, but the value specified by ***lpszDigest*** does not match the computed hash value of files, the function will still return zero.

This function performs the comparison in a way to minimize potential disk I/O whenever possible. First, the size of the two files are compared and if they are different, no further checks are performed. Next, if either the COMPARE_FILE_TIME_CREATED or COMPARE_FILE_TIME_MODIFIED options have been specified, the file timestamps are compared and no further checks are performed if they do not match. A byte comparison of both files will only be performed after the other comparison criteria has been satisfied.

Comparing the contents of a file against a message digest value is the most expensive operation in terms of memory and processor utilization. The larger the files are, and the more complex the hash algorithm, the more compute-intensive the operation will be. In most cases, the SHA-1 algorithm provides a good balance between resource utilization and minimizing the possibility of hash collisions where different files could potentially generate the same hash value.

If it is critically important to minimize the possibility of a hash collision, use the SHA-256 algorithm; however, be aware of the impact it can have on performance when used with very large files. The current thread will block while the hash is being computed and if a large amount of data is being processed, this may cause the application to become non-responsive. In that case, it is recommended you create a background worker thread and call this function from within that worker thread, rather than inside the main UI thread.

To avoid potential problems with relative pathing in multithreaded applications, it is recommended you use absolute paths whenever possible. If you do not specify an absolute path, the file locations will be considered relative to the current working directory for the process. Because this is shared between all threads, it is possible for one thread to change the current working directory at any time and this can create inconsistent results. If you must use relative paths in a multithreaded application, use a mutex or critical section to ensure other threads cannot change the current working directory for the process while performing the file comparison.

This function will normalize the file names provided by the caller and perform checks to make sure the names do not contain illegal characters. If the name includes quotes, wildcard characters or other invalid symbols the function will fail and return zero with the last error code set to ST_ERROR_INVALID_FILE_NAME. It is permitted to use forward slashes in path names and these will be converted to the standard backslash character used with Windows paths. This function cannot be used

to compare the contents of NTFS alternate data streams.

This function is defined using an alias which automatically selects an ANSI or Unicode version based on the value of the UNICODE preprocessor macro. If you use the ANSI version of this function, the expanded file names will always be considered UTF-8 encoded strings, regardless of the current locale of the application. If your application uses file names which contain non-ASCII characters, you should always use the Unicode version of the function, even if your project is configured to use a multi-byte character set.

Example

```
// Compare the contents of a file against an SHA-1 hash value
LPCTSTR lpszFile = _T("%USERPOFILE%\\Documents\\Projects\\MyDocument.docx");
LPCTSTR lpszSHA1 = _T("541E43FDC612E41C02B770A2FD87EBB4A12EA800");
BOOL bIdentical;

bIdentical = CompareFileEx(
                lpszFile,
                NULL,
                COMPARE_FILE_DIGEST_SHA1,
                lpszSHA1);

// Compare the contents of two files
LPCTSTR lpszFile1 = _T("%USERPROFILE%\\AppName\\ProjectData.db");
LPCTSTR lpszFile2 = _T("%TEMP%\\BackupData.db");

bIdentical = CompareFileEx(
                lpszFile1,
                lpszFile2,
                COMPARE_FILE_CONTENTS,
                NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompareFile, CompressFile, GetMessageDigest, ExpandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CompressBuffer Function

```
BOOL WINAPI CompressBuffer(
   LPVOID lpvInputBuffer,
   DWORD cbInputBuffer
   LPVOID lpvOutputBuffer,
   LPDWORD lpcbOutputBuffer
);
```

The **CompressBuffer** function compresses the contents of the specified buffer.

Parameters

lpvInputBuffer

The buffer which contains the data to be compressed.

cbInputBuffer

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

The buffer which will contain the compressed data. This parameter may be NULL, which can be used to determine the size of the of the resulting compressed data prior to allocating memory for it.

lpcbOutputBuffer

The number of bytes that may be copied into the output buffer. If the ***lpvOutputBuffer*** parameter is not NULL, this must be initialized to a non-zero value. When the function returns, the actual number of bytes of compressed data is returned in this parameter.

Return Value

A non-zero value is returned if the data was successfully compressed. A zero value indicates that the data could not be compressed.

Remarks

This function will use the default Deflate algorithm to compress the contents of the buffer. This is a compression algorithm which combines the LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. It generally provides a good balance between the data compression ratio and system resources used. The compression ratio achieved by the **CompressBuffer** function will depend on the type of data being compressed.

The compressed data can be expanded to its original contents by calling the **ExpandBuffer** function.

Example

```
cbOutputBuffer = 0;
bResult = CompressBuffer(lpInputBuffer,
                         cbInputBuffer,
                         NULL,
                         &cbOutputBuffer);

if (bResult)
{
    lpOutputBuffer = (LPBYTE)LocalAlloc(LPTR, cbOutputBuffer);
    bResult = CompressBuffer(lpInputBuffer,
                             cbInputBuffer,
                             lpOutputBuffer,
```

```
                                &cbOutputBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CompressFile, DecodeFile, EncodeFile, ExpandBuffer, ExpandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CompressFile Function

```
BOOL WINAPI CompressFile(
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile
);
```

The **CompressFile** function compresses the contents of the specified file.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the file to be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name. An error will be returned if this parameter is NULL or a zero-length string.

lpszOutputFile

The name of the file that is to contain the compressed file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name. If this parameter is NULL or a zero-length string, the compression is performed in-place.

Return Value

A non-zero value is returned if the file was successfully compressed. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The compression ratio achieved by the **CompressFile** function depends on the type of file that is being compressed. The compressed file is not stored in an archive format that is recognized by third-party applications such as PKZip or WinZip.

This function will use the default Deflate algorithm to compress the file. This is a compression algorithm which combines the LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. It generally provides a good balance between the data compression ratio and system resources used. You can use the **CompressFileEx** function to specify an alternate compression algorithm if you wish to perform more resource-intensive file compression.

If the output file already exists, it will be replaced with the compressed contents of the input file. If the compression fails for any reason, the output file will be deleted and the last error code will be updated indicating the cause of the failure. If you want to ensure an existing output file will be preserved if an error occurs, you should create a temporary file and use it as the output file. If this function succeeds, then you can rename or move the temporary file using a function like **MoveFileEx**.

It is possible to compress the input file in-place by specifying the ***lpszOutputFile*** parameter as a NULL pointer or zero-length string. In this case, the contents of the input file will be compressed to a temporary file and if the compression succeeds, the original input file will be replaced by the temporary file. If an error occurs, the input file will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompressFileEx, DecodeFile, EncodeFile, ExpandFile, ExpandFileEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CompressFileEx Function

```
BOOL WINAPI CompressFileEx(
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile,
   DWORD dwCompressionType,
   DWORD dwCompressionLevel,
   LPVOID lpvReserved,
   DWORD dwReserved
);
```

The **CompressFileEx** function compresses the contents of the specified file.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the file to be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name. An error will be returned if this parameter is NULL or a zero-length string.

lpszOutputFile

The name of the file that is to contain the compressed file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name. If this parameter is NULL or a zero-length string, the compression is performed in-place.

dwCompressionType

A numeric value which determines the algorithm that will be used to compress the data. One of the following values may be specified. By default, the Deflate algorithm is used.

Value	Description
COMPRESSION_TYPE_DEFLATE	A compression algorithm that combines the LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. Deflate is widely used by compression software and provides a good balance between the data compression ratio and system resources used. This is the default compression method.
COMPRESSION_TYPE_BZIP2	A compression algorithm that rearranges blocks of data in sorted order and then uses Huffman coding to process different frequencies of data within the block. Burrows-Wheeler compression provides a better compression ratio than the Deflate algorithm, however it requires more resources to perform the compression.
COMPRESSION_TYPE_LZMA	A compression algorithm that combines the LZ77 algorithm for dictionary-based compression with range encoding to efficiently represent repetitive patterns in the data. LZMA can achieve higher compression ratios than Deflate or the Burrows-Wheeler algorithms; however, it

	requires more memory and CPU resources due to the larger dictionary sizes and more complex encoding algorithm.

dwCompressionLevel

A numeric value which specifies the compression level to use. A value of zero specifies that the default compression level appropriate for the selected algorithm should be used, balancing resource usage and the compression ratio of the data. A value of 1 specifies that the compression should be performed using minimal memory resources, at the expense of the compression ratio. The maximum value of 9 specifies that the algorithm should use more memory to achieve the maximum compression ratio. It is recommended that most applications use the default value of zero.

lpvReserved

A reserved parameter which should always be NULL.

dwReserved

A reserved unsigned integer value which should always be zero.

Return Value

A non-zero value is returned if the file was successfully compressed. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The compression ratio achieved by the **CompressFileEx** function depends on the type of file that is being compressed. The compressed file is not stored in an archive format that is recognized by third-party applications such as PKZip or WinZip.

If the output file already exists, it will be replaced with the compressed contents of the input file. If the compression fails for any reason, the output file will be deleted and the last error code will be updated indicating the cause of the failure. If you want to ensure an existing output file will be preserved if an error occurs, you should create a temporary file and use it as the output file. If this function succeeds, then you can rename or move the temporary file using a function like **MoveFileEx**.

It is possible to compress the input file in-place by specifying the ***lpszOutputFile*** parameter as a NULL pointer or zero-length string. In this case, the contents of the input file will be compressed to a temporary file and if the compression succeeds, the original input file will be replaced by the temporary file. If an error occurs, the input file will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompressBuffer, CompressFile, DecodeFile, EncodeFile, ExpandBuffer, ExpandFile, ExpandFileEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ConvertTextFile Function

```
BOOL WINAPI ConvertTextFile(
   LPCTSTR lpszInputFile,
   UINT uInputCodePage,
   LPCTSTR lpszOutputFile,
   UINT uOutputCodePage,
   DWORD dwOptions,
);
```

The **ConvertTextFile** function converts a text file from one code page to another.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the text file to be converted. The file path can include environment variables surrounded by percent symbols and those variables will be automatically expanded. If you do not specify a full path, the file name will be relative to the current working directory. This parameter cannot be NULL.

uInputCodePage

An integer value which specifies the code page of the input file to be converted. A value of zero specifies the current active code page for the process. An invalid code page will cause the function to fail.

lpszOutputFile

A null-terminated string which specifies the name of the text file to be created. The file path can include environment variables surrounded by percent symbols and those variables will be automatically expanded. If you do not specify a full path, the file name will be relative to the current working directory. If the file does not exist, it will be created. If the file does exist, it will be overwritten. It is permitted for the input file and output file names to be the same, in which case the input file will be replaced. This parameter cannot be NULL unless the FILE_CONVERT_VALIDATE option has been specified.

uOutputCodePage

An integer value which specifies the new encoding. A value of zero specifies the current active code page. An invalid code page will cause the function to fail. If the input file is a Unicode text file and the output code page is not Unicode, data may be lost or corrupted in the file.

dwOptions

An unsigned integer value which specifies one or more option flags which will control how the file is converted. It may be one or more of the following values:

Value	Description
FILE_CONVERT_DEFAULT	The default conversion option, only converting between code pages. The end-of-line character sequences in the text file will not be normalized and the file permissions and timestamps will not be preserved.
FILE_CONVERT_UNICODE	The input file will be checked to determine if it contains valid UTf-8 or UTF-16 encoding. If the file does not contain Unicode text, the conversion will default to using the specified code page. If this option is used with FILE_CONVERT_STRICT and the input file is not Unicode

	text, the function will fail. This option is typically used when you know the input file contains Unicode text, but you are unsure which specific encoding it uses. You can specify the input code page as CP_ACP (0) which will cause the conversion to use the current active code page if the input file is not Unicode text.
FILE_CONVERT_NORMALIZE	If the input text file contains UNIX style end-of-line characters (linefeeds), these will be normalized to the Windows end-of-line character sequence (carriage-return and linefeed). This option will have no effect if the file already uses the Windows end-of-line conventions. This option is typically used when it's possible the input file was copied or downloaded from a non-Windows system and you want to ensure it is consistent with other Windows text files.
FILE_CONVERT_PRESERVE	The output file will have the same attributes, file creation and file modification times as the original input file. If this option is not specified, the output file will have default attributes and current time values. If the output file already exists, this option does not preserve the attributes and timestamps of the file being replaced.
FILE_CONVERT_UTF8_BOM	Ensure that a UTF-8 byte order mark (BOM) is written to the beginning of the output file. This option is only used when the input file is converted to UTF-8, otherwise the option is ignored. The use of byte order marks in UTF-8 encoded text files is uncommon and not all text editors may recognize them.
FILE_CONVERT_NO_BOM	Prevent a byte order mark (BOM) from being written to the beginning of the output file whenever it converted to any Unicode encoding. This is the default for UTF-8, so this option does not need to specified if you converting the input file to UTF-8. Using this option when converting files to UTF-16LE or UTF-16BE is supported, but it is not recommended. Without a BOM, text editors may not recognize the contents as being UTF-16 encoded and may consider the file to binary instead of text.
FILE_CONVERT_STRICT	Perform strict checks during the conversion process. This option will cause the conversion to fail if the file contents cannot be converted without error using the specified code page. By default, characters which cannot be converted using the given code page will be substituted with a placeholder symbol. If this option is not used, it can result in potential data loss if you attempt to convert a Unicode text file to an ANSI code page, such as converting UTF-8 to Windows 1252.
FILE_CONVERT_VALIDATE	The conversion will be performed but an output file is not created or replaced. This option can be used to determine if

	a conversion would be successful without affecting the output file. If this option is used, any output file name passed to this function is ignored.

Return Value

A non-zero value is returned if the file could be converted to the specific code page. A value of zero indicates the conversion process failed. Check the value returned by the **GetLastError** function to determine the reason the file conversion was not successful.

Remarks

The **ConvertTextFile** function converts a text file from one character encoding to another using Windows code pages. It supports UTF-8, UTF-16 (little-endian and big-endian), and common Windows ANSI code pages. It can also normalize line endings and preserve file metadata such as creation/modification times and attributes.

The following Unicode encodings are supported:

- UTF-8 (CP_UTF8, 65001)
  A variable-length encoding where ASCII characters (U+0000-U+007F) are stored as single bytes. Non-ASCII characters use 2-4 bytes. UTF-8 is backward-compatible with ASCII and is commonly used with various Internet protocols. Byte order is not an issue, though a byte order mark (BOM) `EF BB BF` is optional.
- UTF-16LE (1200)
  Encodes characters as 2 or 4 bytes using little-endian byte order. Often includes a BOM `FF FE` to indicate endianness. It is the native encoding for Windows Unicode text.
- UTF-16BE (1201)
  Similar to UTF-16LE, but uses big-endian byte order. Typically includes a BOM `FE FF`. This encoding is occasionally used on UNIX-based systems but is rarely used on Windows.

The code pages for UTF-16LE and UTF-16BE are defined by Microsoft, but they cannot be used with the **WideCharToMultiByte** or **MultiByteToWideChar** functions. Normally, they are only supported for use with managed applications. However, this function will recognize them as valid code pages and perform the appropriate conversion. Unlike UTF-8 which has a defined macro `CP_UTF8`, there are no corresponding macros for UTF-16 in the Windows SDK.

A byte order mark (BOM) may help identify the encoding to some text editors but may also cause compatibility issues with tools that are not designed to be aware of Unicode.

- UTF-16LE and UTF-16BE output includes a BOM unless `FILE_CONVERT_NO_BOM` is specified.
- UTF-8 output does not include a BOM unless `FILE_CONVERT_UTF8_BOM` is specified.

The function can normalize line endings in the input file to match Windows conventions (CRLF) using the `FILE_CONVERT_NORMALIZE` option. This is useful when converting files originally created on UNIX (LF) or macOS Classic (CR). If the file already uses the Windows CRLF convention for the end-of-line, this option will be ignored and has no effect on the output.

When `FILE_CONVERT_UNICODE` is used, the function first checks for a BOM to identify UTF-8 or UTF-16 encodings. If no BOM is found, it analyzes the byte patterns near the beginning of the file to detect if the content is valid UTF-8 or contains UTF-16 characteristics. If detection fails, the code page specified by the *uInputCodePage* parameter will be used. The detection of UTF-8 encoded text without a BOM performs checks to ensure it uses correct codepoints and does not have characters

outside the valid range, therefore it should be considered reliable in determining if it is valid UTF-8 text.

For text files using UTF-16 without a BOM, there is no stringent check that can be made to determine which is being used. Instead, the function uses heuristics to guess which is most likely depending on the order of null bytes and character sequences in the text. While it is generally accurate with text files, it is possible the function could mistakenly identify a binary file as being either UTF-16LE or UTF-16BE encoded text.

If the output code page does not normally support all of the characters used with the input text, they will be replaced with placeholder symbols (typically a "?" character) and the conversion will proceed. This is most often the case if the input and output code pages are not compatible (for example, attempting to convert a text file which was created using Greek code page (1253) to Western European (1252). It can also occur if you attempt to convert a Unicode text file to an ANSI code page. In most cases, the only completely safe conversion is when you convert to a Unicode encoding or between two different Unicode encodings.

To prevent potential data loss or corruption of the output file, use the `FILE_CONVERT_STRICT` option. When this option is used, the function fails when:

- An invalid or ambiguous encoding is detected.
- Any characters cannot be represented in the output encoding.
- The file ends with incomplete multi-byte sequences.

For a complete list of Windows code pages, you can refer to the table of code page identifiers in the Windows SDK documentation. The more common code pages are:

Code Page	Description	Encoding Type
65001	UTF-8	Unicode (variable)
1200	UTF-16 Little Endian (UTF-16LE)	Unicode (fixed)
1201	UTF-16 Big Endian (UTF-16BE)	Unicode (fixed)
1252	Western European (Windows Latin 1)	ANSI
1251	Cyrillic (Windows)	ANSI
1250	Central European (Windows)	ANSI
1253	Greek (Windows)	ANSI
1254	Turkish (Windows)	ANSI
1255	Hebrew (Windows)	ANSI
1256	Arabic (Windows)	ANSI
1257	Baltic (Windows)	ANSI
1258	Vietnamese (Windows)	ANSI
874	Thai (Windows)	ANSI
932	Japanese (Shift-JIS)	DBCS
936	Simplified Chinese (GB2312)	DBCS

949	Korean (Unified Hangul Code)	DBCS
950	Traditional Chinese (Big5)	DBCS

In addition to the code pages listed above, you can also use 0 as a code page. This is defined in the Windows SDK as **CP_ACP** which refers to the system's current ANSI code page. It should only be used when explicitly relying on the default behavior for the local system. The code pages marked as DBCS (Double Byte Character Set) are commonly used for East Asian languages and support a mixture of single- and multi-byte sequences.

Example

```
// Convert a UTF-16 text file to UTF-8
LPCTSTR lpszInputFile = _T("%USERPROFILE%\\Documents\\WideTextFile.txt");
LPCTSTR lpszOutputFile = _T("%USERPROFILE%\\Documents\\TextFile.txt");

BOOL bConverted = ConvertTextFile(
                      lpszInputFile, 1200,
                      lpszOutputFile, CP_UTF8,
                      FILE_CONVERT_DEFAULT);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FilterUnicodeText, GetFileCodePage, IsUnicodeConfusable, IsUnicodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateCustomGuid Function

```
BOOL WINAPI CreateCustomGuid(
    LPGUID lpGuid,
    DWORD dwValue
);
```

The **CreateCustomGuid** function creates a custom GUID that embeds a 32-bit application-defined value.

Parameters

lpGuid

A pointer to a GUID structure which will contain the generated GUID when the function returns. This parameter cannot be NULL.

dwValue

A 32-bit unsigned integer value to embed in the generated GUID. This value is stored in the `Data1` field of the GUID and can represent a counter, identifier, or application-defined metadata. If this value is zero, a random 32-bit value will be used instead.

Return Value

A non-zero value is returned if the function was successful. A return value of zero indicates an error. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The **CreateCustomGuid** function generates a unique 128-bit value that conforms to the Version 8 UUID format as described in RFC 9562. Version 8 UUIDs are designed to support application-specific customization while preserving the structural format of standard UUIDs, including proper version and variant bits.

Unlike Version 1 and Version 7 UUIDs, which encode timestamps, Version 8 UUIDs allow the application to define the semantics of part or all of the UUID content. This implementation stores the 32-bit ***dwValue*** in the `Data1` field of the GUID, followed by 96 bits of cryptographically secure random data. This provides a balance between custom application meaning and guaranteed uniqueness.

If ***dwValue*** is set to zero, the function will use a randomly generated 32-bit value to avoid producing UUIDs that begin with all-zero bits, which may be interpreted as placeholders or uninitialized values in some systems. To obtain the random value from a GUID which was previously generated by this function, use the **GetCustomGuidValue** function.

This function is particularly useful for:

- Generating UUIDs that embed object or session identifiers.
- Encoding a partition key, process ID, or user-defined code for categorization or routing.
- Maintaining backward compatibility with systems that require uniqueness but also need embedded metadata.

When a custom value is specified, only the first 32 bits of the UUID are application-defined. The remaining 96 bits are cryptographically random, making collisions extremely unlikely. The probability of a collision after generating ***n*** GUIDs with the same value is approximately:

$$p \approx n^2 / 2^{97}$$

After generating one million GUIDs with the same custom value, the probability of a collision would still be less than 1 in 10^{20}. In practice, even if the same 32-bit custom value is embedded millions or billions of times, the risk of a collision is negligible.

If you need to create a unique value that uses more than 96 bits of entropy, or you prefer to create a unique token value which is not formatted as a GUID, use the **CreateUniqueToken** function. This function allows you to specify the number of bits of entropy used to create a cryptographically secure token and the algorithm used to encode the randomized data.

As with other UUID formats, the terms GUID (Globally Unique Identifier) and UUID (Universally Unique Identifier) refer to the same 128-bit data structure. While Microsoft traditionally uses the term GUID in Windows development, the terms are interchangeable in both specification and practice.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateStringFromGuid, CreateUniqueToken, GetCustomGuidValue, GetFileTimeFromGuid, GetLocalMachineGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateGuidFromString Function

```
BOOL WINAPI CreateGuidFromString(
    LPCTSTR lpszGuid,
    LPGUID lpGuid
);
```

The **CreateGuidFromString** function parses a string and returns of a GUID value.

Parameters

lpszGuid

A null-terminated string which represents the GUID. This parameter cannot be NULL or a zero-length string. See the remarks below for more information about how the string is parsed.

lpGuid

A pointer to a GUID structure which will contain the parsed GUID in binary form when the function returns. This parameter cannot be NULL. The structure members will be initialized to zero when the function is called and if the string can be parsed, will be updated with the GUID values when it returns. If the function fails, this will contain a GUID_NULL value.

Return Value

A non-zero value is returned if the string contains a valid GUID. A return value of zero indicates the string could not be parsed. For more information, call the **GetLastError** function.

Remarks

The **CreateGuidFromString** function parses a string to determine if it contains a valid GUID value. If it does, it will populate a GUID structure with the appropriate values. This function differs from the **UuidFromString** function which is part of the Windows RPC API and will not accept a NULL pointer to represent a GUID_NULL value. In the context of an RPC call, a missing UUID might be considered to be a default or unspecified value; however, this function is intended to parse and validate the string and performs stricter checks on the input.

The typical format for a GUID string would be:

```
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

When parsing the string representation of the GUID, it will ignore leading and trailing whitespace. If the GUID is enclosed in braces or brackets, the value will be extracted from within them and other characters in the string will be ignored. The hexadecimal values in the string may be separated by a dash or spaces; however, it is not required for the GUID to include separators. For example, all of these string values can be parsed:

```
"550e8400-e29b-41d4-a716-446655440000"
"550E8400 e29b 41d4 a716 446655440000"
"550E8400E29B41D4A716446655440000"
"This string {550e8400-e29b-41d4-a716-446655440000} contains a GUID"
```

If the GUID is enclosed in brace characters `{}` or bracket characters `[]` it must include both the opening and closing braces or brackets. If the GUID is not enclosed in braces or brackets, the string must only contain the GUID value and no extraneous characters other than whitespace.

This function will validate the GUID to ensure it is complete and does not use an invalid version number. You should always check the return value because the function may fail if it's passed a string which generally looks like a valid GUID but does not specify all of the hexadecimal digits (for example,

one too many or few digits) or uses a version number which is outside of the accepted range based on RFC 4122.

Note: The terms GUID (Globally Unique Identifier) and UUID (Universally Unique Identifier) refer to the same 128-bit identifier structure. Microsoft adopted the term GUID early on with the introduction of COM (Component Object Model), and it became the standard terminology in Windows development. Meanwhile, UUID originated from the Open Software Foundation (OSF) and was later standardized by the IETF in RFC 4122. While the names differ, the structure and format are identical, and the terms are used interchangeably in practice.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateHexStringFromGuid, CreateNamespaceGuid, CreateStringFromGuid, GetLocalMachineGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateHexStringFromGuid Function

```
INT WINAPI CreateHexStringFromGuid(
    LPCGUID lpGuid,
    LPTSTR lpszHexString,
    INT nMaxLength
);
```

The **CreateHexStringFromGuid** function creates a hexadecimal representation of the specified GUID.

Parameters

lpGuid

A pointer to a GUID structure. If this parameter is NULL or GUID_NULL the function will return a string with all zeroes.

lpszHexString

A null-terminated string buffer which will contain the hexadecimal representation of the GUID. The buffer must be large enough to include the terminating null character. This parameter cannot be NULL. If the function fails, an empty string will be returned to the caller.

nMaxLength

The maximum number of characters which may be copied into the string buffer. The minimum length for the buffer must be 33 characters. This parameter cannot be zero.

Return Value

If the function succeeds, it returns the number of characters copied into the buffer provided. If the function fails, it will return zero.

Remarks

The **CreateHexStringFromGuid** function returns a string representation of a GUID as a series of hexadecimal digits. Unlike the **CreateStringFromGuid** function, the string returned by this function does not contain any separator characters. If successful, the return value will always be 32 because a GUID represents a 128 byte value.

This function does not validate the GUID structure provided by the caller. For example, it will create a string representation of a GUID which uses a version number which is not valid per RFC 4122. This means it is possible for you to create a string which cannot be parsed by the **CreateGuidFromString** function which does perform checks to ensure the GUID version is valid.

For more information about GUIDs and creating a string which uses the standard notation for GUIDs refer to the **CreateStringFromGuid** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateGuidFromString, CreateNamespaceGuid, CreateStringFromGuid, GetLocalMachineGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateNamespaceGuid Function

```
BOOL WINAPI CreateNamespaceGuid(
    LPCGUID lpNamespace,
    LPCVOID lpvBuffer,
    DWORD dwBufferSize,
    LPGUID lpGuid
);
```

The **CreateNamespaceGuid** function creates a deterministic namespace identifier using the specified GUID.

Parameters

lpNamespace

A pointer to a GUID structure which specifies a valid GUID. This parameter may be NULL or GUID_NULL.

lpvBuffer

A pointer to an arbitrary block of data used to construct the namespace GUID. The contents of the buffer will be hashed using SHA-1 and that value will be combined with the ***lpNamespace*** parameter to create the GUID. This parameter cannot be NULL.

dwBufferSize

An unsigned integer which specifies the number of bytes to be read from the ***lpvBuffer*** parameter. If this value is -1 the function will consider the contents of ***lpvBuffer*** to be a null-terminated UTF-8 encoded string and automatically calculate the length. You should not pass a UTF-16 string as the buffer parameter because the high byte of the character will be interpreted as a terminating null. This value cannot be zero.

lpGuid

A pointer to a GUID structure which will contain the namespace GUID constructed by this function. This parameter cannot be NULL and it will always be initialized to GUID_NULL when the function is called. If the function returns successfully, it will contain a valid Version 5 UUID per RFC 4122.

Return Value

A non-zero value is returned if the function was successful. A return value of zero indicates an error. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

Version 5 UUIDs are deterministic identifiers that are generated by computing a SHA-1 hash of a namespace GUID combined with an arbitrary block of data. The result is a UUID that will always be the same for a given namespace and input value. This makes Version 5 UUIDs ideal for scenarios where consistent identifiers are required for the same logical input, such as mapping a domain name, file path, or object name to a unique ID across different systems or sessions.

The **CreateNamespaceGuid** function adheres to RFC 4122, Section 4.3 and generates UUIDs using the Version 5 algorithm. Unlike Version 4 UUIDs, which are entirely random, Version 5 UUIDs provide stable, reproducible identifiers that are collision-resistant within the specified namespace. By varying the namespace (e.g., for different application subsystems or identity types), you can ensure that the same data produces unique UUIDs within each logical context. This technique is commonly used for consistent resource identifiers, object fingerprints, and mapping of human-readable names to binary IDs.

Although SHA-1 is no longer recommended for cryptographic security purposes, it is still used in Version 5 UUIDs because the UUID standard (RFC 4122) explicitly defines the use of SHA-1 for generating name-based identifiers. In this context, SHA-1 is not used to ensure confidentiality or authentication, but rather to provide a compact and deterministic hash of input data with a very low probability of collision. The 160-bit output of SHA-1 is truncated to 128 bits for use as a UUID, and the version and variant bits are set accordingly. While newer standards may adopt different algorithms (such as SHA-256 in custom UUID formats), SHA-1 remains sufficient and interoperable for RFC 4122 Version 5 UUIDs.

Note: The terms GUID (Globally Unique Identifier) and UUID (Universally Unique Identifier) refer to the same 128-bit identifier structure. Microsoft adopted the term GUID early on with the introduction of COM (Component Object Model), and it became the standard terminology in Windows development. Meanwhile, UUID originated from the Open Software Foundation (OSF) and was later standardized by the IETF in RFC 4122. While the names differ, the structure and format are identical, and the terms are used interchangeably in practice.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateSequentialGuid, CreateStringFromGuid, GetLocalMachineGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateSequentialGuid Function

```
BOOL WINAPI CreateSequentialGuid(
   LPGUID lpGuid,
   ULARGE_INTEGER* lpTimestamp
);
```

The **CreateSequentialGuid** function creates a unique sequential GUID which can be sorted lexicographically.

Parameters

lpGuid

A pointer to a GUID structure which will contain the generated GUID when the function returns. This parameter cannot be NULL.

lpTimestamp

A pointer to a variable that receives the timestamp encoded in the GUID, expressed as the number of milliseconds since the Unix epoch (January 1, 1970 UTC). This parameter is optional and may be NULL if the timestamp is not required.

Return Value

A non-zero value is returned if the function was successful. A return value of zero indicates an error. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The **CreateSequentialGuid** function generates a unique 128-bit value which uses the Version 7 UUID format as described in RFC 9562. It is a new generation method designed to address limitations of older UUID versions. Unlike Version 1 UUIDs, which also embed a timestamp but expose MAC addresses and have coarse time granularity, Version 7 UUIDs are privacy-preserving, forward-compatible, and use millisecond resolution for broader applicability and improved precision.

Version 7 UUIDs are time-based identifiers that encode the number of milliseconds since the Unix epoch (January 1, 1970 UTC) directly into the high-order bits of the UUID. This provides a naturally sortable identifier that preserves creation order and facilitates efficient indexing, logging, and storage in systems that rely on time-based sequencing.

The timestamp portion of a Version 7 UUID represents the number of milliseconds since the epoch, encoded as an unsigned 48-bit integer. This allows for chronological sorting of UUIDs without parsing or inspecting the full binary payload. The remaining bits are randomly generated using a secure system-level random number generator to prevent collisions and ensure entropy across processes, machines, and time intervals.

Version 7 UUIDs are particularly useful for:

- Generating IDs in web APIs, message brokers, and event logs that must reflect creation time.
- Replacing monotonically increasing integers with sortable UUIDs to avoid predictability.
- Designing database indexes that benefit from insertion-order locality without needing sequential integer keys.

The *lpTimestamp* parameter can be used to obtain the the value which corresponds to the encoded time value in the generated GUID. This value can be used for sorting, temporal analysis, or debugging purposes. If you do not require access to the time, this parameter may be set to NULL. To obtain the

timestamp value as a FILETIME structure used by the Win32 API, use the **GetFileTimeFromGuid** function.

Note: Although Version 7 UUIDs are still in the draft standard phase (RFC 9562), they are widely supported by modern tools. The GUIDs produced by **CreateSequentialGuid** conform to the expected structure and semantics defined in the RFC, ensuring forward compatibility with future standards and implementations. This function generates the timestamp using the internal system clock using the highest precision possible. If the system clock is not synchronized with a time server, this may cause time drift in the generated GUID.

As with other UUID formats, the terms GUID (Globally Unique Identifier) and UUID (Universally Unique Identifier) refer to the same 128-bit data structure. While Microsoft traditionally uses the term GUID in Windows development, the terms are interchangeable in both specification and practice.

Example

```
GUID guid = {0};
ULARGE_INTEGER uiTimestamp = {0};

// Create a GUID based on the current date and time
if (CreateSequentialGuid(&guid, &uiTimestamp))
{
    TCHAR szGuid[64] = _T("");
    TCHAR szTime[64] = _T("");
    SYSTEMTIME guidTime = {0};

    CreateStringFromGuid(&guid, szGuid, 64);

    // Get the GUID timestamp as a SYSTEMTIME structure and create
    // a human-readable string; we could also use the GetDateFormat
    // and GetTimeFormat functions in the Win32 API
    //
    // Note the date and time will be in UTC and not localized
    // for the current time zone. The SystemTimeToTzSpecificLocalTime
    // function can convert it to local time

    if (GetSystemTimeFromGuid(&guid, &guidTime))
    {
        wsprintf(szTime, _T("%04d-%02d-%02d %02d:%02d:%02d.%d"),
                 guidTime.wYear,
                 guidTime.wMonth,
                 guidTime.wDay,
                 guidTime.wHour,
                 guidTime.wMinute,
                 guidTime.wSecond,
                 guidTime.wMilliseconds);
    }
    else
    {
        _tprintf(_T("GetSystemTimeFromGuid failed with error 0x%08lX\n"),
GetLastError());
        return;
    }

    // Show the GUID created along with the timestamp value and creation time
    _tprintf(_T("%s (%I64d) created %s\n"), szGuid, uiTimestamp.QuadPart, szTime);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateStringFromGuid, CreateUniqueToken, GetFileTimeFromGuid, GetLocalMachineGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateStringFromGuid Function

```
INT WINAPI CreateStringFromGuid(
    LPCGUID lpGuid,
    LPTSTR lpszGuid,
    INT nMaxLength
);
```

The **CreateStringFromGuid** function creates a string representation of the specified GUID.

Parameters

lpGuid

A pointer to a GUID structure. If this parameter is NULL or GUID_NULL the function will return a string with all zeroes.

lpszGuid

A null-terminated string buffer which will contain the representation of the GUID. The buffer must be large enough to include the terminating null character. This parameter cannot be NULL. If the function fails, an empty string will be returned to the caller.

nMaxLength

The maximum number of characters which may be copied into the string buffer. The minimum length for the buffer must be 37 characters. This parameter cannot be zero.

Return Value

If the function succeeds, it returns the number of characters copied into the buffer provided. If the function fails, it will return zero.

Remarks

The **CreateStringFromGuid** function returns a string representation of a GUID as a series of hexadecimal digits separated into groups using dashes. A GUID (Globally Unique Identifier), also known as a UUID (Universally Unique Identifier), is a 128-bit value used to uniquely identify objects. The structure is defined in RFC 4122 and is widely used in software development for identifiers that must be unique without requiring central coordination. In memory, a GUID is typically represented as a structure composed of four fields: a 32-bit integer (`Data1`), two 16-bit integers (`Data2` and `Data3`), and an array of eight bytes (`Data4`). This layout is consistent with Microsoft's definition in the Windows SDK, and the binary format preserves the required field order and endianness conventions.

GUIDs are commonly serialized into a human-readable string representation, consisting of 32 hexadecimal digits separated by hyphens in the format 8-4-4-4-12, for example:

```
550e8400-e29b-41d4-a716-446655440000
```

The string may optionally be enclosed in braces and the hex digits are case-insensitive. GUID strings are used with Windows in a wide variety of contexts including database keys, COM object identifiers, licensing systems, configuration files, and as identifiers for volumes, devices, or user sessions. The string returned by this function will not be enclosed in braces.

The 128-bit GUID format includes a 4-bit version field that identifies how the value was generated. Commonly used versions include Version 1 (timestamp and MAC address), Version 3 (name-based with MD5 hash), Version 4 (random or pseudo-random), and Version 5 (name-based with SHA-1). Newer drafts introduce Version 6 (time-ordered for improved database indexing), Version 7 (based on Unix timestamps and random bits), and Version 8 (reserved for custom or experimental use). The

version is encoded in the high nibble of the third field (Data3) and provides insight into the origin and structure of the UUID, which can be useful for debugging or ensuring compliance with application-specific requirements.

This function does not validate the GUID structure provided by the caller. For example, it will create a string representation of a GUID which uses a version number which is not valid per RFC 4122. This means it is possible for you to create a string which cannot be parsed by the **CreateGuidFromString** function which does perform checks to ensure the GUID version is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateNamespaceGuid, GetLocalMachineGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CreateUniqueToken Function

```
INT WINAPI CreateUniqueToken(
    LPTSTR lpszToken,
    INT nMaxLength,
    INT nBitLength,
    DWORD dwEncoding
);
```

The **CreateUniqueToken** function generates a unique, cryptographically secure token.

Parameters

lpszToken

A null-terminated string buffer that receives the null-terminated token. The buffer must be large enough to hold the encoded token and its null terminator. If the function fails, an empty string is returned. This parameter is required and cannot be NULL.

nMaxLength

The maximum number of characters that may be copied to the output buffer, including the null terminator. If the buffer is too small to contain the token string, the function will fail.

nBitLength

The number of bits of entropy to use when generating the token. This value must be between 64 and 4096. If the value is not a power of two, it will be automatically rounded up to the nearest power of two. If the value is out of bounds, it will be clamped to the closest valid value. If this parameter is zero, a 128 bit random value will be used to generate the token.

dwEncoding

The encoding algorithm used to convert the randomized data into a printable string. This value must be one of the following encoding constants:

Value	Description
TOKEN_ENCODE_BASE64	Use the Base64 algorithm, a widely supported encoding for binary data. Produces compact output, but includes characters like **+**, **/**, and **=** which may require escaping in URLs or filenames. Useful when interoperability is a priority.
TOKEN_ENCODE_BASE58	Use the Base58 algorithm, designed to generate compact, human-friendly tokens. It excludes ambiguous characters (**`0`**, **`O`**, **`l`**, **`I`**), making it ideal for printed or typed identifiers and safe for use in URLs.
TOKEN_ENCODE_BASE32	Use the Base32 algorithm, which encodes data using 32 unambiguous alphanumeric characters. A good choice for case-insensitive systems or manually entered codes, such as backup codes or license keys.
TOKEN_ENCODE_BASE85	Use the Base85 algorithm for high-density output using the full printable ASCII range. Generates shorter tokens than Base64, but includes punctuation characters, which may not be suitable for all systems without additional escaping.

Return Value

If the function succeeds, it will return the length of the token string not including the terminating null character. If the function fails, it will return a value of zero. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The **CreateUniqueToken** function generates a unique, cryptographically secure token string using the specified encoding format. The token is based on high-entropy random data generated using a secure operating system API and encoded into a compact, printable string. This function is suitable for generating session identifiers, authentication tokens, API keys, or any other unique identifier that must be unguessable and globally unique.

If the ***dwEncoding*** parameter is zero, the function defaults to Base58 encoding, which provides a short, URL-safe, and visually unambiguous format. This is ideal for user-facing or embedded tokens. Each supported encoding format has distinct characteristics:

- **Base64** is compact and compatible with email and web services, but may require escaping for use in URLs or filenames. These tokens can use digits, upper and lower case letters, the **+** and **/** symbols.
- **Base58** avoids ambiguous characters and is well-suited for tokens shown to or entered by users. These tokens use digits and upper and lower case letters without any symbols.
- **Base32** is case-insensitive and easy to read, with predictable output length, making it ideal for printed codes. These tokens only use digits and lower case letters without any symbols.
- **Base85** minimizes token length while remaining printable, but the inclusion of symbols may limit usability in some contexts. These tokens use digits, upper and lower case letters and punctuation characters.

Trailing padding characters (**=**) are automatically removed from Base64 and Base32 encoding to create more compact tokens since decoding is not required.

The ***nBitLength*** value determines the amount of entropy used to generate the token. Values are normalized to the nearest power of two and limited to a maximum of 4096 bits (512 bytes) to ensure performance and memory efficiency. In most cases, a token with 128 bits of entropy is more than sufficient. With 128 bits, over **5.1 quintillion** tokens can be safely generated before the chance of a collision becomes statistically significant.

If you wish to create a token which embeds a specific custom value and uses 96 bits of entropy, you can use the **CreateCustomGuid** function. This function generates a custom 128-bit GUID which embeds an application-defined value and returns a standard version 8 UUID in the standard dash notation.

Example

```
TCHAR szToken[MAX_TOKEN_STRING] = _T("");

// Create a Base32 token using 128 bits of entropy
if (CreateUniqueToken(szToken, MAX_TOKEN_STRING, 128, TOKEN_ENCODE_BASE32))
{
    _tprintf(_T("%s\n"), szToken);
}
else
{
    _tprintf(_T("CreateUniqueToken failed with error 0x%08lX\n"), GetLastError());
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateCustomGuid, CreateNamespaceGuid, CreateSequentialGuid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DecodeBuffer Function

```
INT WINAPI DecodeBuffer(
   LPCTSTR lpszInput,
   LPBYTE lpOutput,
   INT cbOutput,
   DWORD dwOptions
);
```

The **DecodeBuffer** function decodes an encoded string, and stores the result in the specified buffer.

Parameters

lpszInput

A null-terminated string which contains the base64 encoded text.

lpOutput

A pointer to a byte array buffer which is used to store the decoded data. It is recommended that the buffer be as large as the length of the encoded string.

cbOutput

A long integer which specifies the maximum number of bytes which may be stored in the buffer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
DATA_DECODE_BASE64	Use the Base64 algorithm, a widely supported method for encoding binary data as text. It is the standard used by most modern email clients and web applications. This is the default encoding algorithm and should be used by most applications.
DATA_DECODE_QUOTED	Use the quoted-printable algorithm for encoding the data. Most printable characters are left as-is, with control characters and 8-bit characters encoded as their hexadecimal value. Note that this option cannot be combined with the other encoding methods.
DATA_DECODE_URL	Use the URL encoding algorithm for encoding the data. Letters and numbers are left as-is, with control characters, most punctuation and 8-bit characters encoded as their hexadecimal value. Note that this option cannot be combined with the other encoding methods.
DATA_DECODE_BASE58	Use the Base58 algorithm, which produces a compact, human-readable encoded string. Commonly used for short binary identifiers such as cryptocurrency addresses. This algorithm should not be used with large files. Refer to the remarks below for more information about Base58.
DATA_DECODE_BASE32	Use the Base32 algorithm, which encodes binary data into a limited set of 32 alphanumeric characters. Commonly used in systems where case-insensitive encoding is

	required or where data may be transcribed manually. See remarks below for more details about Base32.
DATA_DECODE_BASE85	Use the Base85 algorithm, which encodes binary data into a more compact form than Base64. Useful when file size reduction is critical and full ASCII compatibility is required. See remarks below for more details about Base85.
DATA_DECODE_Z85	Use the Z85 algorithm which is a high-density binary-to-text encoding algorithm designed for use in structured data protocols. It converts 4 bytes (32 bits) of binary data into 5 printable ASCII characters using a compact, fixed character set.
DATA_DECODE_COMPRESSED	The data was compressed prior to being encoded, and should be expanded after the decoding has completed successfully. This option is ignored if the encoding type is not base64. This should only be used if it was specified when the data was encoded.

Return Value

If the function succeeds, it will return the number of bytes decoded and stored in the buffer. If the function fails, it will return -1. Failure typically indicates that the encoded string was corrupted or the buffer is not large enough to store the decoded data.

Remarks

The **DecodeBuffer** function is used to decode a block of data that was previously encoded with the **EncodeBuffer** function. To decode the contents of a file, it is recommended that you use the **DecodeFile** function instead.

To convert UTF-8 encoded text to UTF-16 or localized MBCS (multibyte character set) text, use the **UnicodeDecodeText** function. If you wish to convert the contents of a text file to a different Unicode encoding, use the **ConvertTextFile** function.

Unlike Base64 or Base85, the Z85 encoding algorithm enforces strict alignment. The input string must always be a multiple of 5 characters. It does not use padding and does not support decoding of partial blocks. Any attempt to decode a string whose size is not evenly divisible by 5 will result in an error. This constraint ensures predictable output sizes and simplifies integration with fixed-width protocols.

For more information about the other encoding algorithms, refer to the **EncodeFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, DecodeFile, EncodeBuffer, EncodeFile, GetMessageDigest, IsUnicodeText, UnicodeDecodeText, UnicodeEncodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

DecodeFile Function

```
BOOL WINAPI DecodeFile(
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile,
   DWORD dwOptions,
   DWORD dwReserved
);
```

The **DecodeFile** function opens and decodes an encoded file, storing the contents in the specified file.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the file to be decoded. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

The name of the file that is to contain the decoded file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

dwOptions

An unsigned integer which specifies the decoding method and optional behaviors. The low-word of this parameter specifies the decoding method and the high-word specifies additional options. You may only select one decoding algorithm or the function will fail.

Value	Description
FILE_DECODE_BASE64	Use the Base64 algorithm, a widely supported method for encoding binary data as text. It is the standard used by most modern email clients and web applications. This is the default encoding algorithm and should be used by most applications.
FILE_DECODE_UUCODE	Use the uuencode algorithm, a legacy encoding format traditionally used on UNIX systems and in early email software. This option is primarily included for compatibility with older applications. This algorithm should not be used with new applications.
FILE_DECODE_QUOTED	Use the quoted-printable encoding scheme, which encodes only non-printable characters. Often used in email messages to preserve text readability while ensuring compatibility. This algorithm should only be used when the content is primarily text which may include some 8-bit characters. Do not use this algorithm to encode large amounts of binary data.
FILE_DECODE_YENCODE	Use the yEnc algorithm, a compact encoding method commonly used for binary file attachments in Usenet newsgroups. It offers higher efficiency than traditional

	uuencode or Base64. This algorithm is primarily used with USENET newsgroups.
FILE_DECODE_BASE58	Use the Base58 algorithm, which produces a compact, human-readable encoded string. Commonly used for short binary identifiers such as cryptocurrency addresses. This algorithm should not be used with large files. Refer to the remarks below for more information about Base58.
FILE_ENCODE_BASE32	Use the Base32 algorithm, which encodes binary data into a limited set of 32 alphanumeric characters. Commonly used in systems where case-insensitive encoding is required or where data may be transcribed manually. See remarks below for more details about Base32.
FILE_ENCODE_BASE85	Use the Base85 algorithm, which encodes binary data into a more compact form than Base64. Useful when file size reduction is critical and full ASCII compatibility is required. See remarks below for more details about Base85.
FILE_DECODE_COMPRESSED	Compress the file prior to encoding. This option may be combined with an encoding method to reduce file size. The resulting file must be decompressed after decoding.

dwReserved

This parameter is reserved and should always be set to zero.

Return Value

A non-zero value is returned if the file was successfully decoded. A zero value indicates that the file does not exist, the encoded file was damaged or the output file could not be created.

Remarks

The **DecodeFile** function decodes files that were previously encoded through a call to the **EncodeFile** function or by a third-party application such as an email client. The option to expand a previously compressed file requires that the function be able to create a temporary file on the local system in the directory specified by the TEMP environment variable. This function can only expand a file that was previously compressed with the **EncodeFile** or **CompressFile** functions.

For more information about the encoding algorithms, refer to the **EncodeFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, CompressFile, DecodeBuffer, EncodeBuffer, EncodeFile, ExpandFile, GetMessageDigest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

EncodeBuffer Function

```
INT WINAPI EncodeBuffer(
   LPBYTE lpInput,
   INT cbInput,
   LPTSTR lpszOutput,
   INT cchOutput,
   DWORD dwOptions
);
```

The **EncodeBuffer** function encodes the contents of the specified byte array, converting the data into printable text.

Parameters

lpInput

A buffer that contains the data to be encoded.

cbInput

A long integer which specifies the number of characters in the buffer which should be encoded.

lpszOutput

A null-terminated string that will be used to store the encoded text. The length of the string must be at least 33% larger than the number of bytes being encoded, and will be terminated with a null byte.

cchOutput

The maximum number of characters that may be stored in the string buffer. If this value is too small to store the encoded text, the function will fail.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
DATA_ENCODE_BASE64	Use the Base64 algorithm, a widely supported method for encoding binary data as text. It is the standard used by most modern email clients and web applications. This is the default encoding algorithm and should be used by most applications.
DATA_ENCODE_QUOTED	Use the quoted-printable algorithm for encoding the data. Most printable characters are left as-is, with control characters and 8-bit characters encoded as their hexadecimal value. Note that this option cannot be combined with the other encoding methods.
DATA_ENCODE_URL	Use the URL encoding algorithm for encoding the data. Letters and numbers are left as-is, with control characters, most punctuation and 8-bit characters encoded as their hexadecimal value. Note that this option cannot be combined with the other encoding methods.
DATA_ENCODE_BASE58	Use the Base58 algorithm, which produces a compact, human-readable encoded string. Commonly used for

	short binary identifiers such as cryptocurrency addresses. This algorithm should not be used with large files. Refer to the remarks below for more information about Base58.
DATA_ENCODE_BASE32	Use the Base32 algorithm, which encodes binary data into a limited set of 32 alphanumeric characters. Commonly used in systems where case-insensitive encoding is required or where data may be transcribed manually. See remarks below for more details about Base32.
DATA_ENCODE_BASE85	Use the Base85 algorithm, which encodes binary data into a more compact form than Base64. Useful when file size reduction is critical and full ASCII compatibility is required. See remarks below for more details about Base85.
DATA_ENCODE_Z85	Use the Z85 algorithm which is a high-density binary-to-text encoding algorithm designed for use in structured data protocols. It converts 4 bytes (32 bits) of binary data into 5 printable ASCII characters using a compact, fixed character set.
DATA_ENCODE_COMPRESSED	The data should be compressed before it is encoded. To restore the original data, it must be expanded after it has been decoded. This option is ignored if the encoding type is not base64.
DATA_ENCODE_LINEBREAK	The encoded data should broken into multiple lines of text if the resulting string is longer than 72 characters. This option is ignored if the encoding type is not Base64, Base32 or quoted-printable. This option should be specified if the encoded data is going to be included in an email message.

Return Value

If the function succeeds, it will return the number of bytes encoded and stored in the string. If the function fails, it will return -1. Failure typically indicates that the buffer is not large enough to store the encoded data.

Remarks

The **EncodeBuffer** function is used to encode a block of data and store it in the specified string buffer as printable text. To encode the contents of a file it is recommended that you use the **EncodeFile** function instead.

To convert a string to UTF-8 encoded text, use the **UnicodeEncodeText** function. To convert UTF-8 encoded text to UTF-16 or localized MBCS (multibyte character set) text, use the **UnicodeDecodeText** function. If you wish to convert the contents of a text file to a different Unicode encoding, use the **ConvertTextFile** function.

Unlike Base64 or Base85, the Z85 encoding algorithm enforces strict alignment. The input must be a multiple of 4 bytes and the encoded output is always a multiple of 5 characters. It does not use padding and does not support encoding of partial blocks. Any attempt to encode a buffer whose size is not evenly divisible by 4 will result in an error. This constraint ensures predictable output sizes and simplifies integration with fixed-width protocols.

Z85 is best suited for encoding small binary buffers in memory, such as cryptographic keys, binary hashes, or compact data blocks transmitted in messaging systems. However, Z85 is not appropriate for encoding files unless their size is guaranteed to be a multiple of 4 bytes. Supporting arbitrary-length files would require either padding or a custom wrapper format to store the original length, which would violate the Z85 specification and reduce interoperability. For general-purpose encoding or file storage, Base64 or Base85 is recommended instead.

For more information about the other encoding algorithms, refer to the **EncodeFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, DecodeBuffer, DecodeFile, EncodeFile, GetMessageDigest, IsUnicodeText, UnicodeDecodeText, UnicodeEncodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

EncodeFile Function

```
BOOL WINAPI EncodeFile(
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile,
   DWORD dwOptions,
   DWORD dwReserved
);
```

The **EncodeFile** function opens and encodes a file, storing the contents as printable text in the specified file.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the file to be encoded. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

The name of the file that is to contain the encoded file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

dwOptions

An unsigned integer which specifies the encoding method and optional behaviors. The low-word of this parameter specifies the encoding method and the high-word specifies additional options. You may only select one encoding algorithm or the function will fail.

Value	Description
FILE_ENCODE_BASE64	Use the Base64 algorithm, a widely supported method for encoding binary data as text. It is the standard used by most modern email clients and web applications. This is the default encoding algorithm and should be used by most applications.
FILE_ENCODE_UUCODE	Use the uuencode algorithm, a legacy encoding format traditionally used on UNIX systems and in early email software. This option is primarily included for compatibility with older applications. This algorithm should not be used with new applications.
FILE_ENCODE_QUOTED	Use the quoted-printable encoding scheme, which encodes only non-printable characters. Often used in email messages to preserve text readability while ensuring compatibility. This algorithm should only be used when the content is primarily text which may include some 8-bit characters. Do not use this algorithm to encode large amounts of binary data.
FILE_ENCODE_YENCODE	Use the yEnc algorithm, a compact encoding method commonly used for binary file attachments in Usenet newsgroups. It offers higher efficiency than traditional

	uuencode or Base64. This algorithm is primarily used with USENET newsgroups.
FILE_ENCODE_BASE58	Use the Base58 algorithm, which produces a compact, human-readable encoded string. Commonly used for short binary identifiers such as cryptocurrency addresses. This algorithm should not be used with large files. Refer to the remarks below for more information about Base58.
FILE_ENCODE_BASE32	Use the Base32 algorithm, which encodes binary data into a limited set of 32 alphanumeric characters. Commonly used in systems where case-insensitive encoding is required or where data may be transcribed manually. See remarks below for more details about Base32.
FILE_ENCODE_BASE85	Use the Base85 algorithm, which encodes binary data into a more compact form than Base64. Useful when file size reduction is critical and full ASCII compatibility is required. See remarks below for more details about Base85.
FILE_ENCODE_COMPRESSED	Compress the file prior to encoding. This option may be combined with an encoding method to reduce file size. The resulting file must be decompressed after decoding.

dwReserved

This parameter is reserved and should always be set to zero.

Return Value

A non-zero value is returned if the file was successfully encoded. A zero value indicates that the input file does not exist or the output file could not be created.

Remarks

The **EncodeFile** function is used to convert binary files into text-based representations using one of several supported encoding formats. This is useful when transmitting binary data over protocols that are not 8-bit clean, such as email or text-based APIs. If compression is enabled with FILE_ENCODE_COMPRESSED, the input file will be compressed to a temporary file created in the directory specified by the system's TEMP environment variable. The encoded file is not in a standard archive format and must be decompressed using the **DecodeFile** or **ExpandFile** functions.

If you wish to convert the contents of a text file to a different Unicode encoding, use the **ConvertTextFile** function.

Base64

Base64 is the most commonly used encoding scheme for converting binary data into a text format. It encodes data in blocks of 3 bytes into 4 printable ASCII characters using a well-defined 64-character alphabet. This method is efficient and highly compatible with a wide range of systems, including email clients, web browsers, APIs, and storage systems that expect text input. The Base64 algorithm includes padding to ensure the output is always a multiple of four characters, which helps with alignment and parsing. It is ideal for encoding attachments, configuration files, cryptographic hashes, and any other binary data where compactness and compatibility are important. Base64 is also significantly faster than more complex schemes like Base58, and should be used whenever performance and size efficiency are important, particularly when encoding large files or data streams.

Base58

Base58 is a binary-to-text encoding scheme designed to represent binary data in a compact, human-readable format. It is most commonly known for its use in cryptocurrency applications, particularly Bitcoin addresses. Unlike Base64, which uses a fixed-length representation and padding, Base58 avoids visually similar characters and does not include padding. The Base58 alphabet omits the characters `0` (zero), `O` (capital o), `I` (capital i), and `l` (lowercase L) to reduce the risk of confusion when viewing or entering an encoded string.

Base58 encodes data by interpreting it as a large integer and as a result, the byte order matters. In typical Base58 usage (e.g., Bitcoin), the input is treated as a big-endian number where the most significant byte comes first. This affects how the data is divided and converted to Base58 digits during encoding. However, not all implementations follow this convention. Some online tools or libraries may treat input as little-endian (least significant byte first), leading to different encoded outputs for the same binary data. Both decode to the same binary value and are valid, but they are not interchangeable.

Important: Never use Base58 to encode large files. While Base58 is excellent for encoding small identifiers or hashes, it is not designed for high-performance or large-scale encoding. Base58 encoding is significantly slower than Base64 because it does not operate on fixed-size blocks and requires arbitrary-precision arithmetic with repeated division and modulo operations. If the file is more than a few kilobytes, you should use the Base64 algorithm for encoding.

Base32

Base32 is a text encoding scheme that represents binary data using a set of 32 alphanumeric characters (A–Z and 2–7). It is designed to be case-insensitive, making it suitable for environments where case preservation is unreliable, such as DNS records, QR codes, and certain user-facing identifiers.

Unlike Base64, Base32 uses only uppercase letters and digits, avoiding symbols and padding characters that might be problematic in legacy systems. This makes it easier to transcribe or read aloud, and less prone to corruption when copied manually. However, the tradeoff is reduced efficiency because Base32 requires more characters to encode the same amount of data. Specifically, it encodes every 5 bytes of input as 8 characters of output, resulting in roughly a 60% size increase.

Base32 is useful when binary data must be represented in a constrained character set and the encoded string may be used in contexts that ignore case or disallow punctuation. Examples include secret keys for two-factor authentication (TOTP), unique file identifiers, or human-readable hash values. For general-purpose file encoding, Base64 remains the preferred format due to its higher efficiency and broader adoption.

Base85

Base85 is a high-efficiency binary-to-text encoding method that encodes every 4 bytes of binary data into 5 ASCII characters. Compared to Base64, it provides a more compact representation of binary data, increasing output size by only ~25% rather than 33%. This makes Base85 particularly useful in bandwidth-sensitive applications or when embedding binary data into text files.

The Base85 alphabet includes a wide range of printable ASCII characters, avoiding control characters and whitespace. Unlike Base64, Base85 does not use padding and encodes data in fixed-length 4-byte blocks. Any remaining bytes at the end of the input are encoded using special logic to maintain compactness.

Base85 is an excellent choice when both compactness and full ASCII compatibility are important. It is faster and more efficient than Base58 for larger files, and often preferred over Base64 in advanced document formats and embedded applications. However, due to the expanded character set, it may be less readable and harder to work with manually than Base32 or Base64. For general use, Base64 remains the best compromise between readability, efficiency, and compatibility.

Uuencode / Uudecode

Uuencode (Unix-to-Unix encode) is a legacy encoding format originally developed for transmitting binary files over systems that only supported 7-bit ASCII text, such as early UNIX-based email and file transfer systems. It encodes binary data into lines of printable characters, including a header line that specifies file permissions and a filename. The corresponding uudecode tool can extract and reconstruct the original file from this format.

Although largely obsolete today, uuencode remains useful for compatibility with older systems, embedded environments, and some command-line tools. However, it is less space-efficient than modern alternatives like Base64 and lacks support for Unicode or MIME encoding standards.

Quoted-Printable Encoding

Quoted-printable is a MIME-compliant encoding scheme designed for encoding data that is mostly ASCII text but may contain a few non-printable or special characters. It represents such characters as =XX, where XX is the two-digit hexadecimal value of the byte. Line breaks and spaces at the end of lines are also handled carefully to preserve formatting across different systems. Quoted-printable is ideal for encoding email content in ISO-8859-1 or UTF-8 where text needs to remain mostly readable while ensuring it passes through 7-bit transport layers. It is not efficient for arbitrary binary files, as it may significantly increase the size of the encoded output when many bytes require escaping.

yEnc Encoding

yEnc is a compact binary-to-text encoding format developed as an improvement over uuencode for posting files to Usenet newsgroups. It provides much better space efficiency (only ~1-2% overhead compared to 33% for Base64 or uuencode) and supports 8-bit clean encoding using simple transformations. yEnc files typically include headers and checksums to support multipart files, resume downloads, and error detection. Although it is not a formal Internet standard, yEnc is widely supported by Usenet clients and utilities that deal with large binary file transfers over NNTP. It is best suited for large attachments or downloads over older protocols that favor plain text.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, CompressFile, ConvertTextFile, DecodeBuffer, DecodeFile, EncodeBuffer, ExpandFile, GetMessageDigest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ExpandBuffer Function

```
BOOL WINAPI ExpandBuffer(
   LPVOID lpvInputBuffer,
   DWORD cbInputBuffer
   LPVOID lpvOutputBuffer,
   LPDWORD lpcbOutputBuffer
);
```

The **ExpandBuffer** function expands the contents of the previously compressed buffer.

Parameters

lpvInputBuffer

The buffer which contains the compressed data.

cbInputBuffer

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

The buffer which will contain the expanded data. This parameter may be NULL, which can be used to determine the size of the of the resulting expanded data prior to allocating memory for it.

lpcbOutputBuffer

The number of bytes that may be copied into the output buffer. If the ***lpvOutputBuffer*** parameter is not NULL, this must be initialized to a non-zero value. When the function returns, the actual number of bytes of expanded data is returned in this parameter.

Return Value

A non-zero value is returned if the data was successfully expanded. A zero value indicates that the data could not be expanded. Failure typically indicates that the compressed data was not in a recognized format.

Remarks

The compressed data buffer passed to the **ExpandBuffer** function must have been compressed by either the **CompressBuffer** or **CompressFile** functions. Data read from files that were compressed using third-party utilities such as WinZip will not be recognized by this function.

Example

```
cbOutputBuffer = 0;
bResult = ExpandBuffer(lpInputBuffer,
                       cbInputBuffer,
                       NULL,
                       &cbOutputBuffer);

if (bResult)
{
    lpOutputBuffer = (LPBYTE)LocalAlloc(LPTR, cbOutputBuffer);
    bResult = ExpandBuffer(lpInputBuffer,
                           cbInputBuffer,
                           lpOutputBuffer,
                           &cbOutputBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CompressBuffer, CompressFile, DecodeFile, EncodeFile, ExpandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ExpandFile Function

```
BOOL WINAPI ExpandFile(
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile
);
```

The **ExpandFile** function expands the contents of a previously compressed file.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the file to be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name. An error will be returned if this parameter is NULL or a zero-length string.

lpszOutputFile

The name of the file that is to contain the expanded file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name. If this parameter is NULL or a zero-length string, the compression is performed in-place.

Return Value

A non-zero value is returned if the file was successfully expanded. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The **ExpandFile** function can only expand files that were previously compressed using the **CompressFile** function. It cannot expand the contents of a file stored in an archive format used by third-party applications such as PKZip or WinZip.

This function will use the default Deflate algorithm to expand the contents of the file. If the file was compressed using **CompressFileEx** using anything other than COMPRESSION_TYPE_DEFLATE, the file must be expanded using the **ExpandFileEx** function. This function will fail if a different compression algorithm was used to compress the file.

If the output file already exists, it will be replaced with the expanded contents of the compressed input file. If the input file cannot be expanded for any reason, the output file will be deleted and the last error code will be updated indicating the cause of the failure. If you want to ensure an existing output file will be preserved if an error occurs, you should create a temporary file and use it as the output file. If this function succeeds, then you can rename or move the temporary file using a function like **MoveFileEx**.

It is possible to expand the input file in-place by specifying the ***lpszOutputFile*** parameter as a NULL pointer or zero-length string. In this case, the contents of the input file will be expanded to a temporary file and if the operation is successful, the original compressed input file will be replaced by the temporary file. If an error occurs, the input file will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CompressFile, CompressFileEx, DecodeFile, EncodeFile, ExpandFileEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ExpandFileEx Function

```
BOOL WINAPI ExpandFileEx(
   LPCTSTR lpszInputFile,
   LPCTSTR lpszOutputFile,
   DWORD dwCompressionType,
   LPVOID lpvReserved,
   DWORD dwReserved
);
```

The **ExpandFileEx** function expands the contents of the previously compressed file.

Parameters

lpszInputFile

A null-terminated string which specifies the name of the file which contains the compressed data. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name. An error will be returned if this parameter is NULL or a zero-length string.

lpszOutputFile

The name of the file that is to contain the expanded data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name. If this parameter is NULL or a zero-length string, the compression is performed in-place.

dwCompressionType

A numeric value which determines the algorithm that was used to compress the data. One of the following values may be specified. By default, the Deflate algorithm is used.

Value	Description
COMPRESSION_TYPE_UNKNOWN	The function should attempt to determine what compression algorithm was used when the data was compressed. This is done by examining the header block of the file and checking for certain signatures which can identify the algorithm. If the algorithm cannot be determined, the function will fail. It is recommended that most applications explicitly specify the compression algorithm.
COMPRESSION_TYPE_DEFLATE	A compression algorithm that combines the LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. Deflate is widely used by compression software and provides a good balance between the data compression ratio and system resources used. This is the default compression method.
COMPRESSION_TYPE_BZIP2	A compression algorithm that rearranges blocks of data in sorted order and then uses Huffman coding to process different frequencies of data within the block.

	Burrows-Wheeler compression provides a better compression ratio than the Deflate algorithm, however it requires more resources to perform the compression.
COMPRESSION_TYPE_LZMA	A compression algorithm that combines the LZ77 algorithm for dictionary-based compression with range encoding to efficiently represent repetitive patterns in the data. LZMA can achieve higher compression ratios than Deflate or the Burrows-Wheeler algorithms; however, it requires more memory and CPU resources due to the larger dictionary sizes and more complex encoding algorithm.

lpvReserved

A reserved parameter which should always be NULL.

dwReserved

A reserved unsigned integer value which should always be zero.

Return Value

A non-zero value is returned if the file was successfully expanded. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The **ExpandFileEx** function can only expand files that were previously compressed using the **CompressFile** or **CompressFileEx** functions. It cannot expand the contents of a file stored in an archive format used by third-party applications such as PKZip or WinZip.

The value of the *dwCompressionType* parameter must match the value which was used with the **CompressFileEx** function. This function will fail if you specify a different compression algorithm than what was used to compress the file. Although your application can use COMPRESSION_TYPE_UNKNOWN and attempt to automatically determine how the file was compressed, this is not always reliable.

If the output file already exists, it will be replaced with the expanded contents of the compressed input file. If the input file cannot be expanded for any reason, the output file will be deleted and the last error code will be updated indicating the cause of the failure. If you want to ensure an existing output file will be preserved if an error occurs, you should create a temporary file and use it as the output file. If this function succeeds, then you can rename or move the temporary file using a function like **MoveFileEx**.

It is possible to expand the input file in-place by specifying the *lpszOutputFile* parameter as a NULL pointer or zero-length string. In this case, the contents of the input file will be expanded to a temporary file and if the operation is successful, the original compressed input file will be replaced by the temporary file. If an error occurs, the input file will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

CompressBuffer, CompressFileEx, DecodeFile, EncodeFile, ExpandBuffer, ExpandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FilterHtmlFile Function

```
INT WINAPI FilterHtmlFile(
   LPCTSTR lpszFileName,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **FilterHtmlFile** function filters HTML formatting from the specified file and returns plain text in an output buffer.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file that contains the HTML text to be filtered and normalized. The file must exist and the current process must have read access. The file path may include environment variables surrounded with percent (%) symbols, in which case the variables will be expanded. This parameter cannot be NULL.

lpszBuffer

A null-terminated string buffer which will contain the filtered text. The buffer must be large enough to contain the entire string and a terminating null character or the function will fail. If this parameter is NULL and the ***nMaxLength*** parameter is zero, the function will return the number of characters required to store the filtered text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the ***lpszOutput*** string buffer. If this value is zero, no characters will be copied to the output buffer and the function will return the length of the filtered output, not including the terminating null character. This can be used to determine how much memory should be allocated to store the filtered text.

Return Value

If the function succeeds, it will return the number of characters copied into the output buffer, not including the terminating null character. If the ***lpszBuffer*** parameter is NULL and ***nMaxLength*** is zero, the function will return the number of characters in the filtered text without copying any characters. If the function fails, or there is no text in the input buffer, the function will return zero.

Remarks

The **FilterHtmlFile** function processes a file containing HTML formatted text and returns output suitable for display in text-only environments. HTML tags are removed and entities are converted to their canonical Unicode equivalent. When filtering markup from the text, the function will:

- Remove script, style, and other non-visible or unsafe elements.
- Convert common tag structures, such as lists and headers, into readable plain text.
- Extract and filter hyperlinks and image alt text, ignoring internal links to other sections.
- Remove any embedded credentials in hyperlinks.
- Decode common HTML entities such as `&trade` and `&copy`, as well as numeric entites like `&#x20.`
- Remove all leading and trailing whitespace from the output, including newlines.

Byte Order Marks

A byte order mark (BOM) is a special sequence of bytes at the beginning of a text file that identifies

the encoding used to store the file's contents. In Unicode formats like UTF-8 or UTF-16, this helps software distinguish between different encodings and can also indicate the byte order (big-endian or little-endian). The presence of a BOM will affect how the content is filtered and the following BOMs are recognized:

- UTF-8 BOM (**`EF BB BF`**) the text is processed as UTF-8 and then filtered.
- UTF-16LE BOM (**`FF FE`**) the text is converted to UTF-8 and then filtered.
- UTF-16BE BOM (**`FE FF`**) the text is converted to UTF-16LE, then to UTF-8 and then filtered.

If you call the ANSI version of this function (**FilterHtmlFileA**), the contents will always be returned as UTF-8 encoded text. If you call the Unicode version of the function (**FilterHtmlFileW**), the contents will always be returned as UTF-16LE encoded text. If the file includes a BOM, this will determine how the contents are converted between the different encoding types. If the file does not contain a BOM, it is presumed to contain UTF-8 encoded text.

The function will remove extraneous whitespace within the text, but will attempt to maintain the structure of ordered and unordered lists. It does not parse the style information or Javascript that may be included in the HTML content. If a paragraph uses a style that would normally prevent it from being rendered in a browser (e.g.: the visibility attribute set to "hidden"), it is ignored and the text will be included in the output.

If you are concerned that the HTML may contain obfuscated text or confusable Unicode homoglyphs, you can pass the output from this function to the **FilterUnicodeText** function. However, you should not do this if the original HTML content includes non-Latin characters or symbols because it can cause the text to become unreadable. For example, if the HTML content contains Greek text, those characters may be corrupted during the text normalization process that decomposes the Unicode characters and replaces homoglyphs with standard representations of those characters.

Note: This function returns the number of characters in the filtered text, not including the terminating null. When dynamically allocating memory for the output buffer, make sure you include additional space for the null character. When using the ANSI version of this function, size the *nMaxLength* parameter for the number of UTF-8 bytes required, not the original file size, since UTF-16 to UTF-8 conversion may increase the byte count. If you are unsure as to how much memory is needed, you can pass in *lpszBuffer* as `NULL` and *nMaxLength* as **`0`** to have the function calculate the required size. Be aware that this approach will cause the HTML file to be opened, read and filtered twice.

Most applications export HTML to a file as UTF-8 encoded text. However, some legacy applications may export HTML to text files using a locale-specific character set. In this case, you can use the **ConvertTextFile** function to convert the contents of the file from a specific code page to a Unicode compatible format.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

Example

```
LPCTSTR pszFileName = _T("%USERPROFILE%\\Documents\\TestPage.html");

// Calculate the number of characters required to store the filtered HTML.
// For situations where you know that you're going to be working with
```

```
// relatively small files, it can be more efficient to preallocate a fixed
// size buffer rather than filtering the file twice.

INT cchFiltered = FilterHtmlFile(pszFileName, NULL, 0);
if (cchFiltered > 0)
{
    LPTSTR pszFiltered = (LPTSTR)malloc((cchFiltered + 1) * sizeof(TCHAR));

    if (pszFiltered == NULL)
    {
        // Virtual memory exhausted
        return;
    }

    cchFiltered = FilterHtmlFile(pszFileName, pszFiltered, cchFiltered + 1);
    if (cchFiltered > 0)
    {
        // Write the filtered text to standard output
        _fputts(pszFiltered, stdout);
        fflush(stdout);
    }

    free(pszFiltered);
}
else
{
    // No filtered text, check the last error code for more information
    DWORD dwError = GetLastError();
    _tprintf(_T("Unable to filter \"%s\", error 0x%lx\n"), pszFileName, dwError);
}
```

See Also

ConvertTextFile, FilterHtmlText, FilterMarkdownText, FilterUnicodeText, IsHtmlText, IsMarkdownText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FilterHtmlText Function

```
INT WINAPI FilterHtmlText(
   LPCTSTR lpszInput,
   INT cchInput,
   LPTSTR lpszOutput,
   INT nMaxLength
);
```

The **FilterHtmlText** function filters HTML formatting from the input buffer and returns plain text in an output buffer.

Parameters

lpszInput

A null-terminated string which contains the HTML formatted text that will be filtered and normalized. If the ANSI version of this function is called, the input should be a null-terminated ASCII or UTF-8 encoded string. This parameter cannot be NULL and the string cannot contain embedded null characters.

cchInput

An integer which specifies the number of characters in the input string. If this parameter is -1, the length of the string will be determined by counting the number of characters up to a terminating null character. If the value is less than the length of the input string, only that number of characters will be filtered. If the value is greater than the length of the string, it will be ignored and the entire string will be filtered.

lpszOutput

A null-terminated string buffer which will contain the filtered text. The buffer must be large enough to contain the entire string and a terminating null character or the function will fail. If this parameter is NULL and the ***nMaxLength*** parameter is zero, the function will return the number of characters required to store the filtered text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the ***lpszOutput*** string buffer. If this value is zero, no characters will be copied to the output buffer and the function will return the length of the filtered output, not including the terminating null character. This can be used to determine how much memory should be allocated to store the filtered text.

Return Value

If the function succeeds, it will return the number of characters copied into the output buffer, not including the terminating null character. If the ***lpszOutput*** parameter is NULL and ***nMaxLength*** is zero, the function will return the number of characters in the filtered text without copying any characters. If the function fails, or there is no text in the input buffer, the function will return zero.

Remarks

The **FilterHtmlText** function processes a string containing HTML formatted text, removes the markup and returns output suitable for display in text-only environments. HTML tags are removed and entities are converted to their canonical Unicode equivalent. When filtering markup from the text, the function will:

- Remove all leading and trailing whitespace from the text, including newlines.
- Remove script, style, and other non-visible or unsafe elements.

- Convert common tag structures, such as lists and headers, into readable plain text.
- Extract and filter hyperlinks and image alt text, ignoring internal links to other sections.
- Remove any embedded credentials in hyperlinks.
- Decode common HTML entities such as **&trade** and **&copy**, as well as numeric entites like **&#x20.**

The function will remove extraneous whitespace within the text, but will attempt to maintain the structure of ordered and unordered lists. It does not parse the style information or Javascript that may be included in the HTML content. If a paragraph uses a style that would normally prevent it from being rendered in a browser (e.g.: the visibility attribute set to "hidden"), it is ignored and the text will be included in the output. If there are sections which contain preformatted text, such as text inside **<pre>...</pre>** or **<code>...</code>** tags, individual line and paragraph breaks will be preserved.

If you are concerned that the HTML may contain obfuscated text or confusable Unicode homoglyphs, you can pass the output from this function to the **FilterUnicodeText** function. However, you should not do this if the original HTML content includes non-Latin characters or symbols because it can cause the text to become unreadable. For example, if the HTML content contains Greek text, those characters may be corrupted during the text normalization process that decomposes the Unicode characters and replaces homoglyphs with standard representations of those characters.

This function will not remove Unicode characters from the input text. If the HTML markup includes any entities which are represented as Unicode, such as **½** which represents "½", they will be returned in the output string buffer as Unicode characters. If the ANSI version of this function is called, Unicode will always be returned as UTF-8 encoded characters. If you expect the HTML to include non-ASCII characters, it is recommended that you use the Unicode version of this function.

To determine if the input string is likely to contain HTML markup before filtering it, you can use the **IsHtmlText** function.

If you wish to filter the contents of a text file which contains HTML, you can use the **FilterHtmlFile** function. That function recogizes Unicode byte order marks (BOMs) and will automatically handle any conversion between UTF-8 and UTF-16 based on whether you call the ANSI or Unicode version of the function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, FilterHtmlFile, FilterMarkdownText, FilterUnicodeText, IsHtmlText, IsMarkdownText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FilterMarkdownText Function

```
INT WINAPI FilterMarkdownText(
    LPCTSTR lpszInput,
    INT cchInput,
    LPTSTR lpszOutput,
    INT nMaxLength
);
```

The **FilterMarkdownText** function filters Markdown from the input buffer and returns plain text in an output buffer.

Parameters

lpszInput

A null-terminated string which contains the Markdown text that will be filtered and normalized. If the ANSI version of this function is called, the input should be a UTF-8 encoded string. This parameter cannot be NULL and the string cannot contain embedded null characters.

cchInput

An integer which specifies the number of characters in the input string. If this parameter is -1, the length of the string will be determined by counting the number of characters up to a terminating null character.

lpszOutput

A null-terminated string buffer which will contain the filtered text. The buffer must be large enough to contain the entire string and a terminating null character or the function will fail. If this parameter is NULL and the ***nMaxLength*** parameter is zero, the function will return the number of characters required to store the filtered text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the ***lpszOutput*** string buffer. If this value is zero, no characters will be copied to the output buffer and the function will return the length of the filtered output, not including the terminating null character. This can be used to determine how much memory should be allocated to store the filtered text.

Return Value

If the function succeeds, it will return the number of characters copied into the output buffer, not including the terminating null character. If the ***lpszOutput*** parameter is NULL and ***nMaxLength*** is zero, the function will return the number of characters in the filtered text without copying any characters. If the function fails, or there is no text in the input buffer, the function will return zero.

Remarks

The **FilterMarkdownText** function processes a text string containing Markdown formatting and returns a simplified version with the formatting removed. This preserves the readable content while stripping away syntax such as headings, list markers, emphasis markers, links, and other Markdown elements that are intended for display formatting.

The function works by scanning the text from start to finish and applying a set of rules to recognize common Markdown constructs. For example, headings have their leading hash markers are removed, list markers are normalized to a simple dash or number, emphasis markers such as the asterisk and underscore are stripped, and link text is preserved with the URL shown in parentheses. Code blocks and inline code are retained without the backtick markers. Whitespace is also normalized to make the

output easier to read.

This function is intended for situations where you need plain text output from Markdown input, such as displaying content in a non-HTML context, preparing it for indexing, or passing it to other text-based utilities. It works best when you are confident that the input is actually Markdown; using it on arbitrary text may result in loss of characters that were mistaken for formatting.

The filtering is based on a set of common Markdown patterns and does not attempt to fully parse the Markdown specification. As a result, it may not recognize every possible variation of Markdown and there may be some cases where not all of the formatting is removed.

To determine if the input string is likely to contain Markdown before filtering it, you can use the **IsMarkdownText** function.

Example

```
// Basic Markdown text with an unordered list and hyperlink
LPCTSTR pszInput = _T("# Heading 1\n"
                      "\n"
                      "> A blockquote with *emphasis* and **strong text**.\n"
                      "\n"
                      "- Item 1\n"
                      "- Item 2 with a [link](https://example.com)\n"
                      "\n"
                      "Inline `code` sample.\n");

LPTSTR pszOutput = NULL;
INT cchOutput = 0;

// First, determine how many characters the filtered output will require
cchOutput = FilterMarkdownText(pszInput, -1, NULL, 0);

if (cchOutput > 0)
{
    pszOutput = (LPTSTR)malloc((cchOutput + 1) * sizeof(TCHAR));

    if (pszOutput == NULL)
        return 0; // Handle memory allocation failure

    // Filter out any Markdown in the input string
    cchOutput = FilterMarkdownText(pszInput, -1, pszOutput, cchOutput + 1);

    _tprintf(_T("Original:\n%s\nFiltered:\n%s\n"), pszInput, pszOutput);
    free(pszOutput);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, FilterUnicodeText, IsMarkdownText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FilterUnicodeText Function

```
INT WINAPI FilterUnicodeText(
    LPCTSTR lpszInput,
    INT cchInput,
    LPTSTR lpszOutput,
    INT nMaxLength
);
```

The **FilterUnicodeText** function filters text to replace confusable Unicode homoglyphs, remove symbols and trim whitespace.

Parameters

lpszInput

A null-terminated string which contains the Unicode text that will be filtered and normalized. If the ANSI version of this function is called, the input should be a UTF-8 encoded string. This parameter cannot be NULL and the string cannot contain embedded null characters.

cchInput

An integer which specifies the number of characters in the input string. If this parameter is -1, the length of the string will be determined by counting the number of characters up to a terminating null character.

lpszOutput

A null-terminated string buffer which will contain the filtered text. The buffer must be large enough to contain the entire string and a terminating null character or the function will fail. If this parameter is NULL and the ***nMaxLength*** parameter is zero, the function will return the number of characters required to store the filtered text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the ***lpszOutput*** string buffer. If this value is zero, no characters will be copied to the output buffer and the function will return the length of the filtered output, not including the terminating null character. This can be used to determine how much memory should be allocated to store the filtered text.

Return Value

If the function succeeds, it will return the number of characters copied into the output buffer, not including the terminating null character. If the ***lpszOutput*** parameter is NULL and ***nMaxLength*** is zero, the function will return the number of characters in the filtered text, not including the terminating null. If the function fails, or there is no text in the input buffer, the function will return zero.

Remarks

The **FilterUnicodeText** function filters Unicode text to normalize visually confusable characters, remove symbolic or non-textual code points (such as emojis), and standardize whitespace. It is designed to produce safe, readable output by converting characters to their most recognizable and printable forms.

This function performs multi-stage normalization and filtering of Unicode text, designed to produce clean, printable output from potentially deceptive or malformed input. It applies Compatibility Decomposition (NFKD) to break down complex characters into their canonical equivalents, such as replacing ligatures or accented characters with their base ASCII forms. It also removes Unicode combining marks that could be used to visually obscure or alter text. Leading and trailing Unicode whitespace, including zero-width and non-breaking spaces, is trimmed to improve consistency.

To mitigate spoofing and visual ambiguity, it replaces confusable Unicode homoglyphs with their standard Latin counterparts. For example, the Cyrillic small letter "a" (U+0430) is replaced with the Latin "a" (U+0061). It also detects and maps mathematical alphanumeric symbols and fullwidth characters to their

ASCII equivalents. Non-textual code points, including emojis, pictographs, private-use characters, and deprecated symbols are removed from the input string. This is especially useful in security-sensitive contexts, such as sanitizing domain names, email addresses, usernames, or filenames. It can help prevent spoofing, phishing, and filter evasion through the use of homoglyphs or invisible characters.

If the ANSI version of this function is called, the input string is validated to ensure that it contains only valid UTF-8 encoding. It removes invalid character sequences such as overlong encodings, truncated multibyte sequences, illegal continuation bytes, and code points outside the valid Unicode range. You should never pass a localized ANSI string encoded using the system code page. To normalize such strings, first convert them to UTF-16 using the **MultiByteToWideChar** function, then call FilterUnicodeTextW.

If the Unicode version of this function is called, the input is validated to ensure it contains only well-formed UTF-16. This includes removing unpaired surrogates, invalid code points, unprintable control characters, and private-use characters. If the input includes malformed Unicode, embedded nulls, or arbitrary binary data, it may result in the loss of non-text content during filtering.

This function is deliberately biased toward security rather than readability and can affect the output when applied to text containing legitimate non-Latin characters. It is intended for use on text that is expected to contain only printable plaintext characters such as identifiers, URLs and file paths. For example, it will filter and normalize text containing certain Greek or Cyrillic characters that could be used to spoof ASCII letters or URL delimiters. While this helps mitigate obfuscated or malicious content, it can also make legitimate text in other languages unreadable.

Example

In this example, the input string contains visually confusable Unicode characters from the Mathematical Alphanumeric Symbols block. The **FilterUnicodeText** function maps each character to its plain ASCII equivalent, producing the expected result: "Hello!". This is particularly useful for detecting spoofed identifiers or sanitizing text used in security-sensitive contexts.

```
#ifdef UNICODE
// UTF-16 encoded version of the string "Hello!" using mathematical bold
LPCWSTR pszInput =
L"\xD835\xDC07\xD835\xDC1E\xD835\xDC25\xD835\xDC25\xD835\xDC28\x0021";
#else
// UTF-8 encoded version of the string "Hello!"
LPCSTR pszInput =
"\xF0\x9D\x90\x87\xF0\x9D\x90\x9E\xF0\x9D\x90\xA5\xF0\x9D\x90\xA5\xF0\x9D\x90\xA8\x21";
#endif

LPTSTR pszOutput = NULL;
INT cchOutput = 0;

// First, determine how many characters the filtered output will require
cchOutput = FilterUnicodeText(pszInput, -1, NULL, 0);

if (cchOutput > 0)
{
    pszOutput = (LPTSTR)malloc((cchOutput + 1) * sizeof(TCHAR));

    if (pszOutput == NULL)
        return 0; // Handle memory allocation failure

    // Convert the input to safe, normalized text; should produce "Hello!" in ASCII
    cchOutput = FilterUnicodeText(pszInput, -1, pszOutput, cchOutput + 1);

    _tprintf(_T("The string \"%s\" was filtered to \"%s\"\n"), pszInput, pszOutput);
    free(pszOutput);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, IsUnicodeConfusable, IsUnicodeText, UnicodeDecodeText, UnicodeEncodeText

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetCustomGuidValue Function

```
BOOL WINAPI GetCustomGuidValue(
    LPCGUID lpGuid,
    LPDWORD lpdwValue
);
```

The **GetCustomGuidValue** function returns the 32-bit application-defined value used to create a custom GUID.

Parameters

lpGuid

A pointer to a GUID structure which contains the custom GUID. This parameter must point to a valid v8 UUID and cannot be NULL.

lpdwValue

A pointer to a 32-bit integer that receives the application-defined value used when creating the custom GUID. This parameter cannot be NULL.

Return Value

A non-zero value is returned if the function was successful. A return value of zero indicates an error. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The **GetCustomGuidValue** function extracts the 32-bit application-defined value stored in the `Data1` field of a Version 8 UUID created by the **CreateCustomGuid** function. This value may represent a sequential counter, type identifier, process or session ID, or any other application-specific data encoded at the time the GUID was created. To ensure that the value is extracted from a compatible UUID, this function validates the version number in the GUID to ensure it is a Version 8 UUID as defined in RFC 9562.

This function is intended to be used specifically with GUIDs generated by the **CreateCustomGuid** function, where the first 32 bits (`Data1`) are defined by the application. Version 8 UUIDs are explicitly designed for application-specific use cases, and there is no standardized layout beyond the required version and variant bits. As such, UUIDs generated by other applications or libraries may encode their custom data in different parts of the structure or not at all. Using this function with a Version 8 UUID that was not generated using the same convention may produce meaningless or misleading results.

Because only the first 32 bits of the GUID are application-defined and the remaining 96 bits are created using a cryptographically secure random number generator, the likelihood of a collision between GUIDs that share the same custom value remains extremely small, even when millions or billions of GUIDs are created using the same value. For more information about collision probabilities, refer to the documentation for the **CreateCustomGuid** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateCustomGuid, CreateGuidFromString, CreateNamespaceGuid, CreateSequentialGuid,

CreateStringFromGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetFileCodePage Function

```
UINT WINAPI GetFileCodePage(
   LPCTSTR lpszFileName
);
```

The **GetFileCodePage** function attempts to determine the Windows code page used to create the file.

Parameters

lpszFileName

A null-terminated string which specifies the name of the file to examine. The file path can include environment variables surrounded by percent symbols and those variables will be automatically expanded. If you do not specify a full path, the file name will be relative to the current working directory. The file must exist, and it must be a regular file that can be opened for reading by the current process. This parameter cannot be NULL.

Return Value

A non-zero code page value is returned if the function could sucessfully determine the code page used to create the file. A value of zero indicates that the file could not be accessed, the code page could not be determined or it is not a text file. If the code page cannot be determined, the **GetLastError** function will return `ERROR_NO_UNICODE_TRANSLATION`.

Remarks

The **GetFileCodePage** function is used to determine the Windows code page used to create the text file. This is done by examining the contents of the file and checking for a byte order mark (BOM) which can identify the file. If there is no byte order mark present, it will examine the contents of the file to determine if could be a text file which uses Unicode. The following checks are performed:

- Look for the presence of a UTF-8 BOM or UTF-7 BOM.
- Look for the presence of a UTF-16LE (large endian) or UTF-16BE (big endian) BOM.
- Check if the beginning of the file contains valid UTF-8 encoding with no errors, such as invalid codepoints or overlong encoding sequences.
- Check the contents of the file to determine if there is a regular sequence of null bytes and ASCII character pairs which could indicate UTF-16 without a BOM.

The presence of a byte order marker is the most reliable method for determining if the file contains Unicode text. If there is no BOM at the beginning of the file, the function attempts to infer the possible character encoding based on what byte sequences are found near the beginning of the file. It is important to note that without a BOM, there can be false-positive detections. The following code pages may be returned:

- UTF-7 (CP_UTF7, 65000)
  An obsolete variable-length encoding designed to be more efficient than using quoted-printable encoding in combination with UTF-8. It represents Unicode text as a stream of 7-bit ASCII characters with a shift sequence to switch to 16-bit encoding for characters outside of the normal ASCII range. Byte order is not an issue, but it does have an optional BOM of **2B 2F 76**. This encoding should not be used.
- UTF-8 (CP_UTF8, 65001)

A variable-length encoding where ASCII characters (U+0000-U+007F) are stored as single bytes. Non-ASCII characters use 2-4 bytes. UTF-8 is backward-compatible with ASCII and is commonly used with various Internet protocols. Byte order is not an issue, though an optional BOM of **EF BB BF** may be used.

- UTF-16LE (1200)
  Encodes characters as 2 or 4 bytes using little-endian byte order. Often includes a BOM **FF FE** to indicate endianness. It is the native encoding for Windows Unicode text.

- UTF-16BE (1201)
  Similar to UTF-16LE, but uses big-endian byte order. Typically includes a BOM **FE FF**. This encoding is occasionally used on UNIX-based systems but is rarely used on Windows.

The code pages for UTF-16LE and UTF-16BE are defined by Microsoft, but they cannot be used with the **WideCharToMultiByte** or **MultiByteToWideChar** functions. Normally, they are only supported for use with managed applications. However, this function will recognize them as valid code pages. Unlike UTF-8 which has a defined macro **CP_UTF8**, there are no corresponding macros for UTF-16 in the Windows SDK.

Determining the ANSI code page of a text file by inspecting its contents is fundamentally unreliable because ANSI encodings lack a definitive signature, such as a byte order mark (BOM), to identify which specific code page was used. Unlike Unicode formats like UTF-8 or UTF-16, which may include a BOM or follow strict encoding rules, ANSI code pages are legacy single-byte or multibyte encodings that reuse the same byte values for different characters depending on the locale.

For instance, the byte 0xE1 maps to completely different character in Windows-1252 than it does in Windows-1251. Because many ANSI encodings overlap in the 0x80-0xFF range and are not self-describing, guessing the correct code page without additional context, such as file origin, user locale, or application metadata, is speculative and error-prone. Misidentifying the code page can lead to incorrect character rendering or data corruption. If this function cannot reliably determine the code page, it will return a code page of 0, which can either be interpreted as failure or you could attempt to render the text using the current active code page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, IsUnicodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetFileNameUrl Function

```
INT WINAPI GetFileNameUrl(
   LPCTSTR lpszFileName,
   LPTSTR lpszFileUrl,
   INT nMaxLength,
   DWORD dwOptions
);
```

The **GetFileNameUrl** function creates a file URL based on the name of a local file or network share.

Parameters

lpszFileName

A null-terminated string which specifies the local file name or network share. Although relative paths are supported, they are not recommended and will be automatically converted to absolute paths in the file URL. It is not necessary for the file to exist on the local system.

lpszFileUrl

A null-terminated string buffer which will contain the file URL when the function returns. If this parameter is NULL, the ***nMaxLength*** parameter should have a value of zero. In this case, the function will ignore this parameter and return the length of the URL.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. This must include space for the terminating null character. If the maximum length is not large enough to store the entire URL, the function will fail. If the value of this parameter is zero, the function will return the number of characters required to store the URL, not including the terminating null.

dwOptions

A numeric value which determines how the file URL will be created. It may be one or more of the following values:

Value	Description
FILE_URL_DEFAULT	The file URL will be composed with default options. The case will be preserved and environment variables will not be expanded; instead they will be encoded as part of the URL. This is the same as using a value of zero (e.g.: no options).
FILE_URL_LCASE	The file URL will be returned as all lower-case letters. If this option is not specified, the case of the file as specified in ***lpszFileName*** will be preserved. This option is primarily for cosmetic consistency; file names are not case-sensitive on Windows.
FILE_URL_EXPAND	If the local file name contains any environment variables surrounded by percent (%) symbols, those variables will be expanded before composing the file URL. If this option is not specified, environment variables will be encoded as part of the URL.

Return Value

If the function succeeds, it returns the number of characters copied into the string buffer, not including the terminating null character. If you pass a value of zero as the ***nMaxLength*** parmeter, the function

will return the length of the string without copying any data. If you use this to calculate how much memory to allocate for the file URL, remember to add one to the return value for the terminating null. If the function fails, it will return zero. Call **GetLastError** to retrieve extended error information.

Remarks

The **GetFileNameUrl** function is designed to generate file URLs that conform to RFC 8089, which defines the file:// URL scheme. This can be useful for referencing local files or network shares in a standardized format that can be used in web browsers, log files, or when interoperating with systems that expect a file URL. Relative paths are automatically normalized to absolute paths based on the current working directory. The function also encodes special characters as required by the URL specification (e.g., spaces become %20). UNC paths such as `\\server\share\file.txt`) are converted into file URLs like `file:////server/share/file.txt`.

If the file name contains Unicode characters outside of the standard ASCII range, they will be included as encoded UTF-8 characters. This ensures interoperability with other platforms and allows safe transport over networks using protocols which may not support 8-bit characters. Note the encoding will always use UTF-8, regardless of whether or not you use the Unicode or ANSI (MBCS) version of this function.

File URLs do not interpret percent symbols as environment variable markers by default. Instead, the percent sign (%) is a reserved character in URL syntax and must be encoded as %25. For example, if the file path you provide contains **`%TEMP%`**, this function will encode it as **`%25TEMP%25`** in the URL. If the FILE_URL_EXPAND option is specified, environment variables will be expanded prior to encoding using the current process environment. Internally, this function uses the **ExpandEnvironmentStrings** function, which follows standard Windows conventions.

Although relative paths are supported, their use is strongly discouraged. The function internally calls **GetFullPathName**, which resolves relative paths based on the process's current working directory. This value is shared across all threads in a process and can change at any time, leading to unpredictable results or subtle bugs. For this reason, it is recommended to always provide fully qualified paths to ensure consistent behavior.

This function does not check for the existence of the file. Its sole purpose is to format the path into a valid URL format. If the application requires file validation, it should perform that check separately. If you wish to ensure the file exists, you can either use a function like **GetFileAttributes** (which will return an error if the file or folder does not exist) or you can provide the resulting file URL to the **IsValidFileUrl** function and have it perform the check.

Example

```
LPCTSTR pszFileName = _T("%PUBLIC%\\Documents\\MyProject.docx");
TCHAR szFileUrl[MAX_FILE_URL_LENGTH] = _T("");

// Expand the environment variables in the file name and return a
// file URL which conforms to the RFC 8089 standard
//
// It is recommended that file names be absolute rather than
// relative to the current working directory

INT nLength = GetFileNameUrl(
                    pszFileName,
                    szFileUrl,
                    MAX_FILE_URL_LENGTH,
                    FILE_URL_EXPAND | FILE_URL_LCASE);

if (nLength > 0)
```

```
{
    // Returns the URL file:///c:/users/public/documents/myproject.docx
    _tprintf(_T("The file URL is \"%s\"\n"), szFileUrl);
}
else
{
    _tprintf(_T("Unable to create a file URL for \"%s\"\n"), pszFileName);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetUrlFileName, IsValidFileUrl, NormalizeUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetFileTimeFromGuid Function

```
BOOL WINAPI GetFileTimeFromGuid(
   LPCGUID lpGuid,
   LPFILETIME lpFileTime
);
```

The **GetFileTimeFromGuid** function gets the timestamp from a sequential GUID and returns the value in a FILETIME structure.

Parameters

lpGuid

A pointer to a GUID structure which will contain the sequential GUID. This parameter must point to a valid v7 UUID and cannot be NULL.

lpFileTime

A pointer to a variable that receives the timestamp encoded in the GUID and converts it to a FILETIME structure. This parameter is required and cannot be NULL.

Return Value

A non-zero value is returned if the function was successful. A return value of zero indicates an error. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The **GetFileTimeFromGuid** function extracts the timestamp stored in a Version 7 UUID and populates a FILETIME structure with the value. The timestamp is stored as the number of milliseconds which have elapsed since January 1, 1970 UTC. This function converts that time value to a FILETIME structure which represents the number of 100-nanosecond intervals that have elapsed since January 1, 1601 UTC. When a sequential GUID is generated, it will typically use the system clock to create the timestamp value. If the clock was not synchronized with a time server when the GUID was created, this may cause time drift in the generated GUID.

This function requires a valid GUID which conforms to a Version 7 UUID as defined in RFC 9562 and the timestamp value embedded in the GUID must specify a legitimate time. Internally, the function will permit a certain amount of clock skew to compensate for systems where the system clock is not synchronized. However, if the timestamp represents a time more than 12 hours into the future, this function will return FALSE. Note that in this case, the FILETIME structure will contain the invalid timestamp and the last error code will be set to ST_ERROR_INVALID_TIMESTAMP.

You can use the **GetSystemTimeFromGuid** function to return the time in a SYSTEMTIME structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateNamespaceGuid, CreateSequentialGuid, CreateStringFromGuid, GetSystemTimeFromGuid, GetTimeFromGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetLocalMachineGuid Function

```
BOOL WINAPI GetLocalMachineGuid(
    LPGUID lpGuid
);
```

The **GetLocalMachineGuid** function returns a unique identifier for the local machine.

Parameters

lpGuid

A pointer to a GUID structure which contains the unique identifier for the local machine. This parameter cannot be NULL.

Return Value

A non-zero value is returned if a unique GUID could be created for the local machine. A return value of zero indicates an error.

Remarks

The **GetLocalMachineGuid** function generates a persistent, system-specific identifier in the form of a Version 5 UUID. This GUID is intended to remain consistent across reboots and is unique to the local machine. It is generated by reading the GUID associated with the system's boot volume and combining it with a predefined namespace using a SHA-1 hash. As a result, the returned GUID will typically remain unchanged unless the system is reconfigured to boot from a different volume or the boot drive is reformatted and Windows is reinstalled.

This function can be useful in scenarios where an application needs to determine whether it is running on the same machine that initially created a configuration or data file. By storing the local machine GUID in the configuration file and later comparing it to the current system's identifier, the application can detect if the data was copied to a different system. In such cases, it may choose to revert settings to their defaults or prompt the user for reconfiguration.

If the boot volume GUID cannot be determined, the function will fall back to using the `MachineGuid` value stored in the Windows registry under `HKEY_LOCAL_MACHINE\Software\Microsoft\Cryptography`. While this value is also intended to uniquely identify a Windows installation, it is less reliable than the boot volume GUID because it can be manually modified or duplicated by cloning.

To convert the resulting GUID to a printable string suitable for display or storage, use the **CreateStringFromGuid** function. To compare two GUID string values, use **IsEqualGuidString**, which parses and compares the normalized binary form of the identifiers.

Example

```
GUID guidCurrent = GUID_NULL;
CHAR szGuidFile[MAX_PATH] = "machine.id";
CHAR szStoredGuid[40] = { 0 };
CHAR szCurrentGuid[40] = { 0 };

// Check if the stored GUID file exists
if (PathFileExistsA(szGuidFile))
{
    FILE* fp = fopen(szGuidFile, "r");
    if (fp != NULL)
    {
```

```
        if (fgets(szStoredGuid, sizeof(szStoredGuid), fp) != NULL)
        {
            // Remove newline if present
            szStoredGuid[strcspn(szStoredGuid, "\r\n")] = '\0';
        }
        fclose(fp);
     }
}

// Get the current machine GUID
if (GetLocalMachineGuid(&guidCurrent))
{
    // Convert current GUID to string
    if (CreateStringFromGuidA(&guidCurrent, szCurrentGuid, sizeof(szCurrentGuid)))
    {
        if (!IsEqualGuidStringA(szStoredGuid, szCurrentGuid))
        {
            // The GUIDs are different; likely copied from another system
            // so take appropriate action, such as reverting settings
        }
        else
        {
            // The GUIDs match, this is the same machine
        }

        // If no stored GUID exists yet, save the current one
        if (szStoredGuid[0] == '\0')
        {
            FILE* fp = fopen(szGuidFile, "w");
            if (fp != NULL)
            {
                 fprintf(fp, "%s\n", szCurrentGuid);
                 fclose(fp);
            }
         }
     }
}
else
{
     // Failed to get current machine GUID
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateNamespaceGuid, CreateStringFromGuid, IsEqualGuidString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetMessageDigest Function

```
BOOL WINAPI GetMessageDigest(
   LPVOID lpvBuffer,
   DWORD cbBuffer,
   DWORD dwDigestOptions,
   LPTSTR lpszDigest,
   INT nMaxLength
);
```

The **GetMessageDigest** function computes a message digest for a buffer, string or data file.

Parameters

lpvBuffer

The data that will be processed by the message digest algorithm. This parameter may be NULL, in which case the standard null hash value will be returned. Depending on the options specified, this may point to a block of memory, a null-terminated string or the name of a data file.

cbBuffer

The number of bytes of data that will be processed by the message digest algorithm. If this value is zero, then the standard null hash value will be returned. If the ***lpvBuffer*** parameter is NULL, this value must be zero. If this value is -1 then ***lpvBuffer*** is considered to be a pointer to a null-terminated string.

dwDigestOptions

A numeric value which specifies the algorithm that will be used to generate the message digest, and may be combined with other option flags. The algorithm may be one of the following values:

Value	Description
MESSAGE_DIGEST_DEFAULT	Use the default message digest algorithm, which is the same as specifying MESSAGE_DIGEST_SHA256
MESSAGE_DIGEST_MD5	Use the MD5 algorithm which generates a 128-bit hash value that is returned as as 32 digit hexadecimal string. This algorithm is typically used as a checksum to verify data integrity, however it is not considered cryptographically secure and hash collision is possible.
MESSAGE_DIGEST_SHA1	Use the SHA1 algorithm which generates a 160-bit hash value that is returned as a 40 digit hexadecimal string. This algorithm is typically used as a checksum to verify data integrity, however it is not considered cryptographically secure.
MESSAGE_DIGEST_SHA256	Use the SHA-256 algorithm which generates a 256-bit hash value that is returned as a 64 digit hexadecimal string. This algorithm is used for data integrity validation and various cryptographic uses, such as password storage and document signing.
MESSAGE_DIGEST_SHA384	Use the SHA-384 algorithm which generates a 384-bit hash value that is returned as a 96 digit hexadecimal string. This is a truncated version of the SHA-512 algorithm that uses

	different initial values.
MESSAGE_DIGEST_SHA512	Use the SHA-512 algorithm which generates a 512-bit hash value that is returned as a 128 digit hexadecimal string. This algorithm is used for data integrity validation and various cryptographic uses, such as password storage and document signing.

The following values may be combined with the algorithm ID to specify options that determine how the message digest is computed and returned to the caller:

Value	Description	
0x10000	MESSAGE_DIGEST_MEMORY	The ***lpvBuffer*** parameter is a pointer to a block of memory and the ***cbBuffer*** parameter specifies how many bytes in the buffer should be used to compute the message digest. This is the default behavior if no other options are specified and the ***cbBuffer*** parameter is not -1.
0x20000	MESSAGE_DIGEST_STRING	The ***lpvBuffer*** parameter is a pointer to a null-terminated string and the ***cbBuffer*** parameter specifies how many characters in the string should be used to compute the message digest. If the ***cbBuffer*** parameter is -1 then all characters up to the terminating null byte will be used.
0x40000	MESSAGE_DIGEST_DATAFILE	The ***lpvBuffer*** parameter is a pointer to a null-terminated string that specifies the name of a data file, and the entire contents of the file will be used to compute the message digest. The ***cbBuffer*** parameter is ignored.
0x80000	MESSAGE_DIGEST_BASE64	The message digest will be returned in ***lpszDigest*** as a base64 encoded value rather than a string of hexadecimal digits. This can be useful for services that use HMAC message digests and require the hash to be in base64 format.

lpszDigest

A null-terminated string that will contain the message digest computed using the contents of the specified buffer. The digest value will consist of upper-case hexadecimal numbers, with the length varying based on the algorithm selected. The string buffer must be large enough to accommodate the entire message digest, including the terminating null character. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that may be copied into the ***lpszDigest*** string buffer, including the terminating null character. This value cannot be zero, and must be large enough to accommodate the complete message digest string value. It is recommended that the string buffer be at least MAX_DIGEST_STRING characters in size.

Return Value

A non-zero value is returned if the message digest was computed and returned in the string buffer provided by the caller. A zero value indicates that the digest could not be computed, or the string

buffer provided by the caller was not large enough to contain the message digest value.

Remarks

The **GetMessageDigest** function accepts an arbitrary block of data, a null-terminated string or a data file and computes a fixed-length value (hash) that can be used to uniquely identify that content. These algorithms are designed to be one-way functions that are deterministic, have a low probability of collision and are difficult or infeasible to reverse. It should be noted that the MD5 and SHA1 algorithms are no longer considered cryptographically secure, however they are still widely used for data integrity validation to protect against accidental or unintended changes.

If the MESSAGE_DIGEST_STRING option is specified and the Unicode version of this function is called, the string will be automatically converted from UTF-16 to UTF-8 encoding, and the UTF-8 encoded text will be used to generate the message digest. If you need to generate a message digest using the UTF-16 encoded version of the string, use the MESSAGE_DIGEST_MEMORY option instead. Keep in mind that UTF-16 encoding represents each character with two bytes, so the buffer length should be twice the number of characters in the Unicode string.

If the MESSAGE_DIGEST_DATAFILE option is specified, the file contents will be processed as binary data with no special consideration given for the various end-of-line conventions used in text files, or any Unicode byte order marks in the file. It is permissible for the file to be empty (zero length), in which case the default null-hash value will be returned based on the algorithm selected by the caller.

The current thread will block while the message digest is being computed. If a large amount of data is being processed, this may cause the application to become non-responsive. In that case, it is recommended that you create a background worker thread and call this function from that worker thread, rather than from the main UI thread.

The length of the message digest string returned by this function will always be the same for a given algorithm, regardless of the size of the data buffer. For example, the length of the message digest for the SHA-256 algorithm will always be 64 characters long. Each byte of the message digest is represented by two upper-case hexadecimal numbers.

It is permitted to provide a NULL or zero-length data buffer to this function, in which case the standard null-hash value will be returned. The same null-hash value will also be returned for a zero-length string or an empty file.

Example

```
// Get a SHA-256 message digest for a string value
LPCTSTR lpszValue = _T("The quick brown fox jumps over the lazy dog");
TCHAR szDigest[MAX_DIGEST_STRING];
BOOL bSuccess;

bSuccess = GetMessageDigest(
            lpszValue,
            (DWORD)-1L,
            MESSAGE_DIGEST_SHA256 | MESSAGE_DIGEST_STRING,
            szDigest,
            MAX_DIGEST_STRING);


// Get an MD5 message digest for the contents of a data file
LPCTSTR lpszFileName = _T("testfile.zip");

bSuccess = GetMessageDigest(
            lpszFileName,
            0,
```

```
MESSAGE_DIGEST_MD5 | MESSAGE_DIGEST_DATAFILE,
szDigest,
MAX_DIGEST_STRING);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

DecodeBuffer, DecodeFile, EncodeBuffer, EncodeFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetTimeFromGuid Function

```
BOOL WINAPI GetTimeFromGuid(
   LPCGUID lpGuid,
   ULARGE_INTEGER* lpTimestamp
);
```

The **GetTimeFromGuid** function returns the 64-bit timestamp value from a sequential GUID.

Parameters

lpGuid

A pointer to a GUID structure which will contain the sequential GUID. This parameter must point to a valid v7 UUID and cannot be NULL.

lpTimestamp

A pointer to an unsigned 64-bit integer which receives the timestamp encoded in the GUID. This parameter is required and cannot be NULL.

Return Value

A non-zero value is returned if the function was successful. A return value of zero indicates an error. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The **GetTimeFromGuid** function extracts the timestamp stored in a Version 7 UUID and and returns it as an unsigned 64-bit integer. The timestamp is stored as the number of milliseconds which have elapsed since January 1, 1970 UTC. When a sequential GUID is generated, it will typically use the system clock to create the timestamp value. If the clock was not synchronized with a time server when the GUID was created, this may cause time drift in the generated GUID.

Although the timestamp is generally equivalent to the **__time64_t** type, is not the same as the value returned by the **_time64** function in the C standard library. The value returned by **_time64** represents the number of seconds which have elapsed since January 1, 1970 UTC. The timestamp value returned by this function represents the number of ***milliseconds*** which have elapsed.

This function requires a valid GUID which conforms to a Version 7 UUID as defined in RFC 9562 and the timestamp value embedded in the GUID must specify a legitimate time. Internally, the function will permit a certain amount of clock skew to compensate for systems where the system clock is not synchronized. However, if the timestamp represents a time more than 12 hours into the future, this function will return FALSE. Note that in this case, the ***lpTimestamp*** parameter will contain the invalid timestamp and the last error code will be set to ST_ERROR_INVALID_TIMESTAMP.

The **GetFileTimeFromGuid** function can be used to obtain the same time value in a FILETIME structure and the **GetSystemTimeFromGuid** function can be used to obtain the time in a SYSTEMTIME structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateNamespaceGuid, CreateSequentialGuid, CreateStringFromGuid, GetFileTimeFromGuid, GetSystemTimeFromGuid, IsEqualGuidString

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetUrlFileName Function

```
INT WINAPI GetUrlFileName(
   LPCTSTR lpszFileUrl,
   LPTSTR lpszFileName,
   INT nMaxLength,
   DWORD dwOptions
);
```

The **GetUrlFileName** function parses a file URL and returns the name of a local file or network share.

Parameters

lpszFileUrl

A null-terminated string which specifies the file URL. It is not necessary for the file URL to reference a file which exists on the local system.

lpszFileName

A null-terminated string buffer which will contain the file name when the function returns. If this parameter is NULL, the ***nMaxLength*** parameter should have a value of zero. In this case, the function will ignore this parameter and return the length of the complete file path.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. This must include space for the terminating null character. If the maximum length is not large enough to store the entire file name, the function will fail. If the value of this parameter is zero, the function will return the number of characters required to store the URL, not including the terminating null.

dwOptions

A numeric value which determines how the file URL will be parsed. It may be one or more of the following values:

Value	Description
FILE_URL_DEFAULT	The file URL will be parsed with default options. The case will be preserved and environment variables will not be expanded. File URLs which contain certain non-standard conventions, such as using the vertical bar (|) as a drive separator will be permitted. It is generally is recommended you use this relaxed parsing mode for the broadest compatibility with most file URLs.
FILE_URL_STRICT	The file URL will be parsed with strict conformance to the standard outlined in RFC 8089. Using this option will cause the function to reject some of the older, non-standard conventions which should no longer be used. You should only use this option if you need to ensure the URLs conform to the current standard.
FILE_URL_LCASE	The file name will be returned as all lower-case letters. If this option is not specified, the case of the path and file name in ***lpszFileUrl*** will be preserved in the file name returned by the function. This option is primarily for cosmetic consistency; file names are not case-sensitive on Windows.
FILE_URL_EXPAND	If the URL contains any encoded environment variables, those

	variables will be expanded before returning the file name. If this option is not specified, environment variables will not be expanded and returned as-is. Note that because environment variables use the same percent symbol that is used for URL encoding, they must be encoded as %25NAME%25.

Return Value

If the function succeeds, it returns the number of characters copied into the string buffer, not including the terminating null character. If you pass a value of zero as the *nMaxLength* parameter, the function will return the length of the string without copying any data. If you use this to calculate how much memory to allocate for the file name, remember to add one to the return value for the terminating null. If the function fails, it will return zero. Call **GetLastError** to retrieve extended error information.

Remarks

The **GetUrlFileName** function parses a file URL and returns the equivalent absolute Windows path. This includes support for URLs that reference local files, mapped drives, and UNC network shares. The returned path is normalized to remove extraneous path delimiters and redundant components (such as . or .. directory references) and will ensure it adheres to the standard Windows path format. The file name returned by this function is always an absolute path.

The function supports both relaxed and strict parsing modes. The default behavior is to be permissive and allow certain legacy formats, such as the use of a vertical bar (|) in place of a colon to separate the drive letter (e.g., file:///C|/path). Strict parsing, enabled with the FILE_URL_STRICT flag, enforces conformance to RFC 8089 and will reject such non-standard forms. This mode is useful when input validation or standards compliance is required.

Although the file URL does not need to refer to an existing file, the returned path will always be fully qualified. If the URL contains encoded environment variables (e.g., **%25TEMP%25**), they will be expanded only if the FILE_URL_EXPAND option is specified. Unlike web URLs, file URLs on Windows are generally case-insensitive; however, the FILE_URL_LCASE flag can be used for cosmetic or consistency purposes when needed.

URL encoding ensures interoperability with other platforms and allows safe transport over networks using protocols which may not support 8-bit characters. If the file URL contains encoded UTF-8 characters they will be automatically decoded. If you use the ANSI version of this function, the file name will always be returned as a UTF-8 encoded string to prevent any possible data loss. If there is any chance the file path will contain Unicode characters, it is recommended you always use the Unicode version of this function.

File URLs do not interpret percent symbols as environment variable markers by default. Instead, the percent sign (%) is a reserved character in URL syntax and must be encoded as %25. For example, to encode **%TEMP%**, you must use **%25TEMP%25** in the URL. If the FILE_URL_EXPAND option is specified, these encoded variables will be expanded using the current process environment. Internally, this function uses **ExpandEnvironmentStrings**, which follows standard Windows conventions.

This function does not validate the file's existence or access permissions, it is solely concerned with converting a URL to a Windows-compatible file path. If validation is required, it must be performed separately after calling this function. If you wish to check that the file exists, you can use the **IsValidFileUrl** function.

It is important to note that file URLs always reference absolute paths. Some older browsers and early versions of the Java runtime supported file URLs with relative paths by omitting the authority portion of the URL; however, this function explicitly prohibits this usage because relative paths may not be

deterministic in multithreaded applications. The current working directory is stored as a global value for each process, and if one thread changes the current working directory, it will change for all other threads and can cause unpredictable behavior.

Example

```
LPCTSTR pszFileUrl = _T("file:///users/public/documents/myproject.docx");
TCHAR szFileName[MAX_PATH] = _T("");

// Return the file name for the specified URL and apply checks to
// ensure the URL conforms to the standard and does not use any
// non-standard conventions common with older browsers

INT nLength = GetUrlFileName(
                    pszFileUrl,
                    szFileName,
                    MAX_PATH,
                    FILE_URL_STRICT);

if (nLength > 0)
{
    // Returns the file C:\users\public\documents\myproject.docx
    _tprintf(_T("The file name is \"%s\"\n"), szFileName);
}
else
{
    _tprintf(_T("Unable to get the file name for \"%s\"\n"), pszFileUrl);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileNameUrl, IsValidFileUrl, NormalizeUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IsEqualGuidString Function

```
BOOL WINAPI IsEqualGuidString(
   LPCTSTR lpszGuid1,
   LPCTSTR lpszGuid2
);
```

The **IsEqualGuidString** function compares two GUID strings and determines if they represent the same binary value.

Parameters

lpszGuid1

A null-terminated string which specifies a GUID. This parameter cannot be NULL.

lpszGuid2

A null-terminated string which specifies a GUID to be compared against the ***lpszGuid1*** parameter. This parameter cannot be NULL.

Return Value

A non-zero value is returned if the two GUIDs represent the same value. A return value of zero indicates they represent different GUIDs or at least one of the string values does not represent a valid GUID.

Remarks

The **IsEqualGuidString** function compares two serialized GUID strings and determines if they represent the same values, even if the strings themselves are not identical. When parsing the string representation of the GUID, this function will ignore leading and trailing whitespace. If the GUID is enclosed in braces or brackets, the value will be extracted from within them and other characters in the string will be ignored. The hexadecimal values in the string may be separated by a dash or spaces; however, it is not required for the GUID to include separators.

Consider these two strings:

```
"{550e8400-e29b-41d4-a716-446655440000}"
"550E8400E29B41D4A716446655440000"
```

Although these strings would be considered different using a standard string comparison function, they represent the same GUID value and the **IsEqualGuidString** function would return TRUE. If you wish to compare a string against a GUID structure, you can use the **CreateStringFromGuid** function to create a string representation of the GUID structure, which you can then pass to this function.

If one or both of the strings represents an invalid GUID, this function will return FALSE even if the strings themselves are identical. For example, if both strings are identical and have the appearance of valid GUIDs but the 4-bit version field in the GUID is invalid, this function will fail because it requires both GUIDs to represent valid values which conform to RFC 4122.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

CreateGuidFromString, CreateHexStringFromGuid, CreateNamespaceGuid, CreateStringFromGuid, GetLocalMachineGuid

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IsHtmlText Function

```
BOOL WINAPI IsHtmlText(
   LPCTSTR lpString,
   INT nLength
);
```

The **IsHtmlText** function checks if a string appears to contain HTML formatted text.

Parameters

lpString

A null-terminated string. This parameter cannot be NULL and the string must be terminated with a null character.

nLength

An integer value which specifies the number of characters to check. If this value is -1, the length of the string is determined by counting the number of characters up to the terminating null character. This parameter cannot be zero or the function will fail with `ERROR_EMPTY`.

Return Value

A non-zero value is returned if the string contains valid HTML. A zero value indicates the string does not appear to contain any HTML formatting. On failure, call the **GetLastError** function to obtain more information about the error.

Remarks

The **IsHtmlText** function examines a text string and uses a set of simple rules to decide whether it appears to contain HTML formatting. It does not fully parse the text; instead it looks for common features such as headings, paragraphs, links and common entity names. Each feature it finds contributes to an internal score, and the function returns a positive result if the score is high enough.

This function is useful when you need to check whether text is likely to contain HTML before passing it to the **FilterHtmlText** function or another function that removes or processes formatting.

Because the detection is based on heuristics, it should be viewed as an educated guess rather than a definitive result. While it is generally accurate for typical HTML content, there may be occasional false positives, particularly for similarly structured text such as XML.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, FilterHtmlText, FilterMarkdownText, IsMarkdownText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IsMarkdownText Function

```
BOOL WINAPI IsMarkdownText(
   LPCTSTR lpString,
   INT nLength
);
```

The **IsMarkdownText** function checks if a string appears to contain text formatted using Markdown.

Parameters

lpString

A null-terminated string. This parameter cannot be NULL and the string must be terminated with a null character.

nLength

An integer value which specifies the number of characters to check. If this value is -1, the length of the string is determined by counting the number of characters up to the terminating null character. This parameter cannot be zero or the function will fail with `ERROR_EMPTY`.

Return Value

A non-zero value is returned if the string contains valid Markdown. A zero value indicates the string does not appear to contain any Markdown formatting. On failure, call the **GetLastError** function to obtain more information about the error.

Remarks

The **IsMarkdownText** function examines a text string and uses a set of simple rules to decide whether it appears to contain Markdown formatting. It does not fully parse Markdown; instead it looks for common features such as headings, lists, blockquotes, links, images, code blocks, and emphasis markers. Each feature it finds contributes to an internal score, and the function returns a positive result if the score is high enough and there is more than one type of Markdown feature present.

The detection process is designed to be fast and to avoid false positives. For example, a single numbered line will not be treated as a list unless other Markdown elements are also present. Special cases such as fenced code blocks are handled so that their contents do not trigger matches for other features.

This function is useful when you need to check whether text is likely to be Markdown before passing it to the **FilterMarkdownText** function or another function that removes or processes formatting. It works best on plain text and UTF-8 encoded strings.

Because the detection is based on heuristics, it should be viewed as an educated guess rather than a definitive result. While it is generally accurate for typical Markdown content, there may be occasional false positives or false negatives depending on the input.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, FilterHtmlText, FilterMarkdownText, IsHtmlText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IsUnicodeConfusable Function

```
BOOL WINAPI IsUnicodeConfusable(
    LPCTSTR lpString,
    INT nLength
);
```

The **IsUnicodeConfusable** function checks if a string contains characters which can be used to obsfucate Unicode text.

Parameters

lpString

A null-terminated string. This parameter cannot be NULL.

nLength

An integer value which specifies the number of characters to check. If this value is -1, the length of the string is determined by counting the number of characters up to the terminating null character. This parameter cannot be zero.

Return Value

A non-zero value is returned if the string contains one or more characters which could be confused with another character. A zero value indicates no confusable characters were found in the string.

Remarks

The **IsUnicodeConfusable** function checks whether the input string contains Unicode characters that may be visually confused with common Latin letters, digits, or symbols. These characters are often used in phishing, obfuscation, or evasion attacks by mimicking familiar text using characters from other scripts (e.g., Cyrillic, Greek) or stylistic variants (e.g., fullwidth or mathematical alphanumerics). This check can be used to flag suspicious input, enforce stricter character policies, or warn users about potentially deceptive content.

For example, a domain name such as `paypal.com` using the Cyrillic 'p' and 'a' characters may appear visually identical to `paypal.com`, but represent a different string entirely. This function detects the presence of such characters by comparing each Unicode code point against an internal map of known homoglyphs.

The function returns a non-zero value if at least one character in the string matches a known confusable code point. It does not modify the string or perform normalization. If normalization is also required, use the **FilterUnicodeText** function instead.

This function is deliberately biased toward security and may return false positives when applied to text containing legitimate non-Latin characters. It is intended for use on text that is expected to contain only printable plaintext characters such as identifiers, URLs and file paths. For example, it will return TRUE if the text contains certain Greek or Cyrillic characters that could be used to spoof ASCII letters or URL delimiters. While this helps detect obfuscated or malicious content, it can also flag legitimate text that appears in a normal message or document.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, FilterUnicodeText, GetFileCodePage, IsMarkdownText, IsUnicodeText, UnicodeDecodeText, UnicodeEncodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IsUnicodeText Function

```
BOOL WINAPI IsUnicodeText(
   LPCTSTR lpString,
   INT nLength
);
```

The **IsUnicodeText** function checks if a string contains structurally valid Unicode text.

Parameters

lpString

A null-terminated string. This parameter cannot be NULL and the string must be terminated with a null character.

nLength

An integer value which specifies the number of characters to check. If this value is -1, the length of the string is determined by counting the number of characters up to the terminating null character. This parameter cannot be zero or the function will fail with `ERROR_EMPTY`.

Return Value

A non-zero value is returned if the string contains valid Unicode characters. A zero value indicates the string contains characters that are not valid Unicode, or the string does not contain any Unicode characters. On failure, call the **GetLastError** function to obtain more information about the error.

Remarks

There are two versions of this function, **IsUnicodeTextA** which checks a multi-byte string to ensure that it contains valid UTF-8 encoded text, and **IsUnicodeTextW** which checks a wide string to ensure it contains valid UTF-16 text.

If the value of the ***nLength*** parameter is larger than the number of characters in the string, the function will not check beyond the terminating null character. If the length of the string is unknown, specify a length of -1 and the function will check the entire contents of the string up to the terminating null character.

If the input string is empty, the function fails with the last error code set to `ERROR_EMPTY` to indicate that the operation requires actual data to validate. This is distinct from `ERROR_INVALID_DATA`, which is used for non-empty input that is structurally malformed.

If a byte-order mark (BOM) is present at the start of the buffer and no additional code units follow, the function still returns success. The BOM itself is considered valid Unicode text.

This function does not perform checks to ensure the string contains printable characters. It only validates that the Unicode string is structurally valid. The **IsUnicodeTextA** function will check to make sure there are no invalid UTF-8 encodings. The **IsUnicodeTextW** function will check to make sure the UTF-16 string does not contain any unpaired surrogates. If you wish to filter the text to include only printable characters, use the **FilterUnicodeText** function.

This function does not preserve state information and cannot be used to check the validity of a stream of text or binary data. It should only be used to validate complete Unicode strings terminated with a null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertTextFile, FilterUnicodeText, GetFileCodePage, IsMarkdownText, IsUnicodeConfusable, UnicodeDecodeText, UnicodeEncodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IsValidFileUrl Function

```
BOOL WINAPI IsValidFileUrl(
   LPCTSTR lpszFileUrl,
   BOOL bFileExists
);
```

The **IsValidFileUrl** function checks if a string represents a valid file URL.

Parameters

lpszFileUrl

A null-terminated string which specifies a file URL. This parameter cannot be NULL or point to a zero-length string.

bFileExists

An integer value which determines if the function checks to make sure the file exists. If this parameter is non-zero, the file must exist or the function will fail. If this parameter is zero, the function will succeed if the URL conforms to the standard, regardless if the file exists on the local system.

Return Value

A non-zero value is returned if the string contains a valid file URL. A zero value indicates the file URL is invalid, or if the ***bFileExists*** parameter is non-zero, the file cannot be found on the local system.

Remarks

A file URL is a special kind of Uniform Resource Identifier (URI) that uses the **file://** scheme to identify local or network files. The format is standardized in RFC 8089, but real-world usage, particularly on Windows, introduces many quirks and compatibility issues. The **IsValidFileUrl** function validates that a given string conforms to the general structure of a file URL, accounting for both the official standard and the numerous platform-specific variations commonly encountered on Windows systems. It operates in a relaxed mode by design, allowing for:

- Drive letter paths like `file:///c:/path/file.txt`
- UNC paths like `file://server/share/path/file.txt`
- Localhost and machine name authorities like `file://localhost/path/file.txt` or `file://server/share/file.txt`
- Encoded environment variables like `file:///%25TEMP%25/file.txt`
- Use of legacy syntax like C| instead of C: for drive letters
- Implicit authority names and administrative shares like `file://C$/path`

Unlike stricter parsers, this function permits more forgiving syntax to accommodate how file URLs are often generated or copy-pasted from browser address bars, log files, or third-party tools. It also automatically expands environment variables if present, allowing URLs like `file:///%25UserProfile%25/Documents/file.txt` to resolve correctly.

In addition to syntax ***bFileExists*** parameter can be used to verify that the file path resolves to an actual file on the local system or accessible network share. This is useful in scenarios where file URLs are user-provided or come from untrusted sources, such as configuration files or external input.

Validating file URLs helps prevent incorrect or malicious input, reduces the risk of referencing special device names (such as **CON**, **NUL** or **PRN**), and ensures that other functions receive well-formed URLs

that will behave as expected. To create a file URL from a local file name, you can use the **GetFileNameUrl** function which will return the complete file path as a valid URL. To parse file URLs and resolve them to a local file name, you can use the **GetUrlFileName** function.

It is important to note that file URLs always reference absolute paths. Some older browsers and early versions of the Java runtime supported file URLs with relative paths by omitting the authority portion of the URL; however, this function explicitly prohibits this usage because relative paths may not be deterministic in multithreaded applications. The current working directory is stored as a global value for each process, and if one thread changes the current working directory, it will change for all other threads and can cause unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileNameUrl, GetUrlFileName, NormalizeUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NormalizeUrl Function

```
INT WINAPI NormalizeUrl(
   LPCTSTR lpszUrl,
   DWORD dwFlags,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **NormalizeUrl** function parses and canonicalizes a URL into a safe, navigable form. It can enforce the scheme, validate and normalize the host name, collapse and encode the path and query parameters as needed, and optionally convert local file paths to `file://` URLs.

Parameters

lpszUrl

A null-terminated string which specifies the URL to be normalized. This parameter cannot be NULL or an empty string.

dwFlags

Specifies one or more bit flags that control how the URL is normalized. A value of zero specifies that only standard normalization will be performed, such as removing any leading and trailing whitespace and ensuring the URL does not contain any illegal characters. Multiple flags may be combined using a bitwise or operator.

Value	Description
URL_FLAG_DEFAULT_HTTPS	When no URL scheme is provided and no scheme is otherwise required, select `https://` by default. If an explicit port is present, port 443 selects `https` and any other port selects `http`. Explicit input schemes are preserved.
URL_FLAG_REQUIRE_HTTP	When no scheme is provided, force `http://`. If the input already specifies a scheme, it is preserved.
URL_FLAG_REQUIRE_HTTPS	When no scheme is provided, force `https://`. If the input already specifies a scheme, it is preserved.
URL_FLAG_REJECT_LOCALHOST	Reject URLs that specify `localhost` as the host name.
URL_FLAG_REJECT_IPV4	Reject URLs that specify a numeric IPv4 address (for example, `192.0.2.1`) as the host.
URL_FLAG_REJECT_IPV6	Reject URLs that specify a bracketed IPv6 address (for example, `[2001:db8::1]`) as the host.
URL_FLAG_ASCII_HOSTNAME	Host names are checked to ensure that they only contain legal ASCII characters. Unicode host names are converted to ASCII (punycode) before validation, and the host name is normalized to lowercase.
URL_FLAG_ESCAPE_PATH	Percent-encode the path component as needed. Non-ASCII characters are encoded as UTF-8 and then percent-encoded per byte. Valid `%HH` sequences are preserved and upper-cased. The path is also normalized by removing duplicate slashes and collapsing `.` and `..` segments of the path.

URL_FLAG_ESCAPE_QUERY	Percent-encode the query and fragment tail as needed. The first **?** is treated as the start of the tail and characters commonly used in queries (such as **&**, **=**, **+**, **;**, **#**) are permitted unescaped. Non-ASCII characters are encoded as UTF-8 and then percent-encoded.
URL_FLAG_VALIDATE_UTF8	Applicable to the ANSI version only. Treat the input as UTF-8 and fail if it is not valid UTF-8. When this flag is not specified, the ANSI version attempts normalize the URL as UTF-8 first and falls back to using the system ANSI code page if the URL is not valid UTF-8.
URL_FLAG_REJECT_LOCAL_FILES	Reject absolute local and UNC paths and existing **file://** URLs. When specified, UNC paths are also rejected and unsafe file extensions are not allowed.
URL_FLAG_REJECT_UNC_PATHS	When local file normalization is permitted, reject UNC paths (for example, **\\server\share**). Files paths which use a mapped drive to a network share will be allowed.
URL_FLAG_ALLOW_UNSAFE_FILES	When converting local/UNC paths to **file://** URLs or normalizing existing **file://** URLs, permit any file extension. If this option is not specified, only a common set of safe, renderable file types are allowed. This includes HTML files, text files and common image formats.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the function returns. This parameter may be NULL, in which case the function returns the number of characters required to store the normalized URL. If a buffer is provided, it must be large enough to contain the entire URL.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. If this parameter is zero, the function will return the number of characters required to store the normalized URL and no characters will be copied into the buffer.

Return Value

An integer which specifies the length of the normalized URL, not including the terminating null character. A return value of zero indicates the URL could not be normalized, either due to an invalid URL or the output buffer not being large enough to store the complete URL. Call the **GetLastError** function for more information about the cause of the failure.

Remarks

The Unicode version validates the input as well-formed UTF-16. Leading and trailing whitespace is trimmed before processing, and an internal maximum is enforced to guard against excessively large inputs. The ANSI version attempts to decode the input as UTF-8 first and falls back to the active ANSI code page unless **URL_FLAG_VALIDATE_UTF8** is specified. When possible, it is recommended that you use the Unicode version of this function to ensure support for internationalized domain names. If the URL is being used for display purposes, it's not always necessary to have the function encode the host name using punycode. However, if you are passing the URL to another application or logging the URL to a text file, it's recommended that you use the **URL_FLAG_ASCII_HOSTNAME** flag to ensure the host

name is converted to a printable ASCII text format.

When local file paths are permitted, absolute drive paths such as `C:\path\file.txt` and UNC paths such as `\\server\share\file.txt` are converted to `file://` URLs. Drive paths are canonicalized, any long-path prefix is removed, backslashes are converted to forward slashes, dot-segments are resolved, and spaces are percent-encoded. If you are normalizing a URL entered by a user and want to prevent them from being able to specify a local file path, use the `URL_FLAG_REJECT_LOCAL_FILES` flag. To permit any local file type, use the `URL_FLAG_ALLOW_UNSAFE_FILES` flag. Normally you should only specify this flag if you want to check the file extension in your application rather than using the internal list of safe file types. Opening local file URLs which specify script files or executables can present a significant security risk.

Backslash characters in a path are normalized to forward slashes before parsing and any illegal characters are rejected. The input string may be a full URL with `http://` or `https://`, or scheme-relative URLs starting with `//`. The authority portion is parsed as `host[:port]` with checks for bracketed IPv6 addresses and strict port validation. This function accepts `localhost`, IPv4, and bracketed IPv6 addresses unless explicitly rejected by flags. When the `URL_FLAG_ASCII_HOSTNAME` bitflag is set, Unicode host names are always converted to IDNA (punycode) before validation and the host name is converted to lower case.

When a host name is present and the path is empty, a single `/` is inserted. The path is always normalized by removing duplicate slashes and resolving `.` and `..` segments without moving above the preserved root. When `URL_FLAG_ESCAPE_PATH` is set, Unicode characters are encoded as UTF-8 and then percent-encoded per byte; valid `%HH` pairs are preserved and uppercased, and `/` is kept as a separator. For the query and fragment portion of the URL, the `URL_FLAG_ESCAPE_QUERY` flag ensures characters are percent-encoded as needed, allowing the `&`, `=`, `+`, `;`, and `#` characters to be preserved in the URL unescaped. Note that this function will always consider the hashmark character to be the beginning of a URL fragment and will not encode it.

Scheme selection preserves any URL scheme explicitly provided by the caller. Otherwise, `URL_FLAG_REQUIRE_HTTP` or `URL_FLAG_REQUIRE_HTTPS` forces the scheme. If neither is specified and a port is present, port 443 selects `https` and any other port selects `http`. For example, the string `"example.com"` would be normalized as `http://example.com/` by default. In most cases, it is recommended you specify the `URL_FLAG_DEFAULT_HTTPS` flag to ensure host names entered without a scheme use a secure connection on port 443 rather than a standard (insecure) connection on port 80.

On success, the function sets the last error to `NO_ERROR`. Common failures include `ERROR_INVALID_PARAMETER` for malformed inputs or illegal characters, `ERROR_NO_UNICODE_TRANSLATION` for invalid UTF-16 (Unicode version) or invalid UTF-8 when validation is requested (ANSI version), and `ERROR_INSUFFICIENT_BUFFER` when the output buffer is too small. This function will never return a partial URL.

This function is primarily intended for validating and normalizing user input. Although is can be used to convert a local file path to a local file URL, the **GetFileNameUrl** function is more flexible and permits the use of environment variables in the local file path. The **GetUrlFileName** function can be used to convert a file URL back to a local file name.

Example

```
LPCTSTR pszUrlList[] = {
    _T("example.com"),
    _T("//Example.COM/../a/./b?q=1#frag"),
    _T("C:\\Docs\\Read Me.txt"),
    _T("file:///C:/Docs/Read%20Me.txt"),
```

```
    _T("about:Blank")
};

// A collection of common flags used for safe user input
const DWORD dwFlags =
    URL_FLAG_DEFAULT_HTTPS    |
    URL_FLAG_ASCII_HOSTNAME   |
    URL_FLAG_ESCAPE_PATH      |
    URL_FLAG_ESCAPE_QUERY;

for (int nIndex = 0; nIndex < ARRAYSIZE(pszUrlList); ++nIndex)
{
    TCHAR szNormalized[1024];

    if (NormalizeUrl(pszUrlList[nIndex], dwFlags, szNormalized,
_countof(szNormalized)))
    {
        _tprintf(_T("OK:     %s\n"), szNormalized);
    }
    else
    {
        DWORD dwError = GetLastError();
        _tprintf(_T("FAILED: %s (%lu)\n"), pszUrlList[nIndex], dwError);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetFileNameUrl, GetUrlFileName, IsValidFileUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

UnicodeDecodeText Function

```
INT WINAPI UnicodeDecodeText(
   UINT nCodePage,
   LPCSTR lpUtf8Text,
   INT nLength,
   LPTSTR lpString,
   INT nMaxLength
);
```

The **UnicodeDecodeText** function decodes UTF-8 encoded text and returns the contents in a string buffer.

Parameters

nCodePage

A value that specifies the code page used when decoding the text. This parameter can be set to the value of any code page that is available in the operating system. It is recommended that you use CP_ACP which specifies the system default ANSI code page. This parameter is only used when decoding UTF-8 encoded text to a multi-byte string.

lpUtf8Text

A null-terminated string that contains the UTF-8 encoded text to be decoded. This parameter must always specify a pointer to an 8-bit character string, regardless if the project is configured to use the Unicode or Multi-Byte character set. This parameter cannot be NULL.

nLength

The number of bytes in the UTF-8 encoded string to be decoded. If this parameter is -1, the length of the UTF-8 encoded text is determined by counting the number of characters up to the terminating null character.

lpString

A null-terminated string buffer that will contain the decoded UTF-8 text when the function returns. The string buffer must be large enough to contain all of the encoded text, and cannot be a NULL a pointer.

nMaxLength

The maximum number of characters that can be copied into the string buffer. The contents of the ***lpString*** buffer will always be null-terminated, and the maximum size must be large enough to include the null character. This value must be greater than zero.

Return Value

If the UTF-8 encoded text is successfully decoded, the return value is the number of characters copied to the string buffer. If the text cannot be decoded, or the string buffer is not large enough to store all of the decoded text, the function will return zero. To get extended error information, call **GetLastError**.

Remarks

There are two versions of this function, **UnicodeDecodeTextA** which returns a localized multi-byte string and **UnicodeDecodeTextW** which converts the UTF-8 encoded text to UTF-16 text. Your project configuration typically determines which version of this function is used by default.

If the value of the ***nLength*** parameter is larger than the number of UTF-8 characters, the function will not check beyond the terminating null character. If the length of the ***lpUtf8Text*** string is unknown,

specify a length of -1 and the function will decode the entire contents of the string up to the terminating null character.

When calling **UnicodeDecodeTextA** to convert UTF-8 encoded text to a localized multi-byte string, it is recommended that you specify CP_ACP (zero) as the code page value unless you know it contains Unicode characters that cannot be represented using the default ANSI code page. Using CP_ACP will ensure the UTF-8 text is decoded using the current locale and language settings.

When calling **UnicodeDecodeTextW** to convert UTF-8 encoded text to UTF-16 text, the code page parameter is ignored and should always be a value of zero.

This function performs a strict check on the UTF-8 encoded text and will fail if the encoding is malformed, or in the case of being converted to a multi-byte string, if it cannot be decoded using the specified code page. It will not simply replace invalid character sequences with a default character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsUnicodeText, UnicodeEncodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

UnicodeEncodeText Function

```
INT WINAPI UnicodeEncodeText(
   UINT nCodePage,
   LPCTSTR lpString,
   INT nLength,
   LPSTR lpUtf8Text,
   INT nMaxLength
);
```

The **UnicodeEncodeText** function encodes a string and returns UTF-8 encoded text.

Parameters

nCodePage

A value that specifies the code page used when encoding the text. This parameter can be set to the value of any code page that is available in the operating system. It is recommended that you use CP_ACP which specifies the system default ANSI code page. This parameter is only used when encoding a multi-byte string.

lpString

A null-terminated string that contains the text to be encoded. This parameter cannot be NULL.

nLength

The number of characters in the string to be encoded. If this parameter is -1, the length of the string is determined by counting the number of characters up to the terminating null character.

lpUtf8Text

A pointer to a character buffer which will contain the UTF-8 encoded text when the function returns. This parameter must always specify a pointer to an 8-bit character string, regardless if the project is configured to use the Unicode or Multi-Byte character set. This parameter cannot be NULL.

nMaxLength

The maximum number of bytes that can be copied into the UTF-8 text buffer. The contents of the ***lpUtf8Text*** buffer will always be null-terminated, and the maximum size must be large enough to include the null byte. This value must be greater than zero.

Return Value

If the string is successfully encoded, the return value is the number of characters copied to the output buffer. If the text cannot be encoded, or the output buffer is not large enough to store all of the encoded text, the function will return zero. To get extended error information, call **GetLastError**.

Remarks

There are two versions of this function, **UnicodeEncodeTextA** which converts a multi-byte string to UTF-8 encoded text, and **UnicodeEncodeTextW** which converts a UTF-16 string to a UTF-8 string. Your project configuration typically determines which version of this function is used by default.

If the value of the ***nLength*** parameter is larger than the number of characters in ***lpString***, the function will not check beyond the terminating null character. If the length of ***lpString*** is unknown, specify a length of -1 and the function will encode the entire contents of the string up to the terminating null character.

When calling **UnicodeEncodeTextA** to convert a multi-byte string to UTF-8 encoding, it is recommended that you specify CP_ACP (zero) as the code page value unless you know it contains

ANSI characters from a different code page. Using CP_ACP will ensure the string is encoded using the current locale and language settings.

This function performs a strict check on the multi-byte input string and will fail if it contains a malformed multi-byte sequence or characters that cannot be converted to UTF-8 using the specified code page. It will not simply replace invalid character sequences with a default character.

When calling **UnicodeEncodeTextW** to convert UTF-16 text to UTF-8 encoded text, the code page parameter is ignored and should always be a value of zero. The text will be normalized prior to being converted to UTF-8 using canonical composition, where decomposed characters are combined to create their canonical precomposed equivalent.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csncdv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IsUnicodeText, UnicodeDecodeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Client Library

Transfer files between a local and server and perform common file management functions on the server.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSFTPV12.DLL
Version	12.0.3024.1286
LibID	A86C8154-EC75-4C39-BB05-B698A87E98F1
Import Library	CSFTPV12.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

The File Transfer Protocol (FTP) library provides a comprehensive API which supports both high level operations, such as uploading or downloading files, as well as a collection of lower-level file I/O functions. In addition to file transfers, an application can create, rename and delete files and directories, search for files using wildcards and perform other common file management functions.

Files can be stored on the local file system or in memory, depending on the needs of your application and multiple file transfers be performed using a single function call. The library can also be used to manage files on the server and supports many of the common protocol extensions that can be used to access the remote file system. It understands a number of different directory listing formats, including those typically used with UNIX and Linux based systems, Windows servers and several other platforms.

This library supports active and passive mode file transfers, firewall compatibility options, proxy servers and secure file transfers using TLS 1.3 and SFTP (SSH 2.0). Secure file transfers support implicit and explicit TLS sessions, client certificates and 256-bit AES encryption. SFTP transfers support public/private key authentication using either RSA or OpenSSH formats.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must

have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Functions

Function	Description
FtpAddQueuedFile	Add a new file to the queue to be uploaded or downloaded.
FtpAddQueuedFileEx	Add a new file to the queue with additional transfer options.
FtpAllocate	Allocate the specified number of bytes on the server.
FtpAsyncConnect	Connect asynchronously to the specified server.
FtpAsyncFileList	Return an unparsed list of files in the specified directory.
FtpAsyncGetData	Copy the contents of a remote file to a local buffer.
FtpAsyncGetFile	Copy a file from the server to the local system.
FtpAsyncGetFileEx	Copy a file from the server to the local system, supporting files larger than 4 GiB.
FtpAsyncProxyConnect	Connect asynchronously to the specified proxy server.
FtpAsyncPutData	Create a file on the server using the contents of a local buffer.
FtpAsyncPutFile	Copy a file from the local system to the server.
FtpAsyncPutFileEx	Copy a file from the local system to the server, supporting files larger than 4 GiB.
FtpAttachThread	Attach the specified client handle to another thread.
FtpCancel	Cancel the current blocking operation.
FtpCancelQueuedFile	Cancel the queued file transfer.
FtpChangeDirectory	Change the current working directory on the server.
FtpChangeDirectoryUp	Change the current working directory on the server.
FtpClearQueue	Remove all files from the current transfer queue.
FtpCloseDirectory	Close the open directory on the server.
FtpCloseFile	Close the current file on the server.
FtpCommand	Send a command to the server.
FtpConnect	Establish a client connection with a server.
FtpConnectUrl	Establish a client connection using the specified URL.
FtpCreateDirectory	Create the specified directory on the server.
FtpCreateQueue	Create a new file transfer queue.
FtpCreateSecurityCredentials	Create a new security credentials structure.
FtpDeleteFile	Delete a file from the server.
FtpDeleteQueue	Delete a previously created file transfer queue.
FtpDeleteSecurityCredentials	Delete a previously created security credentials structure.
FtpDisableEvents	Disable event notification.
FtpDisableTrace	Disable logging of network function calls.
FtpDisconnect	Disconnect the client session from the server.
FtpDownloadFile	Download a file from the server to the local system.
FtpEnableEvents	Enable event notification.
FtpEnableFeature	Enable the specified feature in the client.
FtpEnableTrace	Enable logging of network function calls to a file.

FtpEndOfFile	Determine if the end-of-file has been reached.
FtpEnumFiles	Return a list of files in a directory on the server.
FtpEnumFilesEx	Return a list of files in a directory on the server, supporting files larger than 4 GiB.
FtpEnumQueuedFiles	Return a list of files in the transfer queue.
FtpEnumTasks	Return a list of asynchronous tasks.
FtpEventProc	Callback function that processes events generated by the client.
FtpFindQueuedFile	Find a file in the transfer queue which matches the search criteria.
FtpFreezeEvents	Suspend and resume event handling by the client.
FtpGetActivePorts	Return the range of local port numbers used for active transfers.
FtpGetAutoFileType	Return the file transfer type based on the file extension or content.
FtpGetBufferSize	Return the size of an internal buffer used during data transfers.
FtpGetChannelMode	Return the mode for the specified communication channel.
FtpGetClientQuota	Return quota information for the current client session.
FtpGetData	Copy the contents of a remote file to a local buffer.
FtpGetDefaultZoneId	Returns the default security zone information for downloaded files.
FtpGetDirectory	Get the current working directory on the server.
FtpGetDirectoryFormat	Get the format which is used by the server to list files.
FtpGetErrorString	Return a description for the specified error code.
FtpGetFeatures	Return the features available to the client.
FtpGetFile	Copy a file from the server to the local system.
FtpGetFileEx	Copy a file from the server to the local system, supporting files larger than 4 GiB.
FtpGetFileList	Return an unparsed list of files in the specified directory.
FtpGetFileNameEncoding	Return the character encoding used when sending a file name to the server.
FtpGetFilePathUrl	Return a normalized URL for the specified file on the server.
FtpGetFilePermissions	Return the access permissions for the specified file.
FtpGetFileSize	Return the size of a file on the server.
FtpGetFileStatus	Return file status information from the server.
FtpGetFileStatusEx	Return file status information from the server, supporting files larger than 4 GiB.
FtpGetFileTime	Return the modification time for the specified file on the server.
FtpGetFileType	Return the default file type for the current session.
FtpGetFileZoneInfo	Return file zone identifier information for the specified local file.
FtpGetFirstFile	Return the first file from the file list returned by the server.
FtpGetFirstFileEx	Return the first file from the file list returned by the server, supporting files larger than 4 GiB.
FtpGetFirstQueuedFile	Return information about the first file in the transfer queue.
FtpGetIdleTimeout	Return the idle timeout period for the control channel.
FtpGetLastError	Return the last error code.
FtpGetMultipleFiles	Copy multiple files from the server to the local system.
FtpGetNextFile	Return the next file from the file list returned by the server.
FtpGetNextFileEx	Return the next file from the file list returned by the server, supporting files larger than 4 GiB.

FtpGetNextQueuedFile	Return information about the next file in the transfer queue.
FtpGetPriority	Return the current priority for file transfers.
FtpGetProxyType	Return the proxy type selected by the client.
FtpGetQueueClient	Return the handle to the current client session.
FtpGetQueuedFile	Return information about a file queued for transfer.
FtpGetQueueStatus	Return the status of the current file transfer queue.
FtpGetQueueThread	Return a value which uniquely identifies the queue manager thread.
FtpGetResultCode	Return the result code from the previous command.
FtpGetResultString	Return the result string from the previous command.
FtpGetSecurityInformation	Return security information about the current client connection.
FtpGetSecurityZoneId	Returns the security zone identifier for the specified resource.
FtpGetServerInformation	Get system information about the server.
FtpGetServerStatus	Return system status of server.
FtpGetServerTimeZone	Return the timezone offset in seconds for the current server.
FtpGetServerType	Return the type of operating system the server is running on.
FtpGetStatus	Return the current client status.
FtpGetTaskError	Return the last error code for the specified asynchronous task.
FtpGetTaskId	Return the unique task identifier associated with the specified client session.
FtpGetText	Download the contents of a text file to a string buffer.
FtpGetThreadQueue	Return the handle to a queue associated with the specified thread.
FtpGetTimeout	Return the number of seconds until an operation times out.
FtpGetTransferStatus	Return current file transfer statistics.
FtpGetTransferStatusEx	Return current file transfer statistics, supporting files larger than 4 GiB.
FtpInitialize	Initialize the library and validate the specified runtime license key.
FtpIsBlocking	Determine if the current operation is blocked.
FtpIsConnected	Determine if the client is connected to the server.
FtpIsReadable	Determine if the client can read data from the data channel.
FtpIsWritable	Determine if the client can write data to the data channel.
FtpLogin	Login to the server.
FtpLogout	Logout from the server.
FtpNormalizePath	Normalize the specified file name returning a canonical path.
FtpNormalizeUrl	Normalize the specified URL returning a canonical path.
FtpMountStructure	Mount a structure (filesystem) on the server.
FtpOpenDirectory	Open the specified directory for reading.
FtpOpenFile	Open the specified file for reading or writing.
FtpProxyConnect	Establish a connection with a proxy server.
FtpPutData	Create a file on the server using the contents of a local buffer.
FtpPutFile	Copy a file from the local system to the server.
FtpPutFileEx	Copy a file from the local system to the server, supporting files larger than 4 GiB.

FtpPutMultipleFiles	Copy multiple files from the local system to the server.
FtpPutText	Create a text file on the server from the contents of a string buffer.
FtpRead	Read data from the server.
FtpRegisterEvent	Register an event handler for the specified event.
FtpRegisterFileType	Associate a file name extension with a specific file type.
FtpRemoveDirectory	Remove a directory from the server.
FtpRemoveQueuedFile	Remove a file from the transfer queue.
FtpRenameFile	Rename a file on the server.
FtpReset	Reset the client connection.
FtpResetQueue	Reset the internal state of file transfers in the specified queue.
FtpResumeQueue	Resume the transfer of files after queue processing has been paused.
FtpSetActivePorts	Set the range of local port numbers used for active transfers.
FtpSetBufferSize	Set the size of an internal buffer used during data transfers.
FtpSetChannelMode	Change the security mode for the specified channel.
FtpSetDefaultZoneId	Set the default security zone for downloaded files.
FtpSetDirectoryFormat	Set the format which is used by the server to list files.
FtpSetFeatures	Set the features which can be used by the client.
FtpSetFileMode	Set the current file mode.
FtpSetFileNameEncoding	Set the character encoding type used when sending a file name to the server.
FtpSetFilePermissions	Set the access permissions for the specified file.
FtpSetFileStructure	Set the current file data structure.
FtpSetFileTime	Set the modification time for the specified file on the server.
FtpSetFileType	Set the default file type for the current session.
FtpSetFileZoneInfo	Set or remove the zone indentifier associated with the specified file.
FtpSetIdleTimeout	Set the idle timeout period for the control channel.
FtpSetLastError	Set the last error code.
FtpSetPassiveMode	Set the server in passive mode.
FtpSetPriority	Set the priority for file transfers.
FtpSetTimeout	Set the number of seconds until an operation times out.
FtpStartQueue	Begin transferring the files in the specified queue.
FtpStopQueue	Stop transferring the files in the specified queue.
FtpSuspendQueue	Pause the transfer of files in the specified queue.
FtpTaskAbort	Abort the specified asynchronous task.
FtpTaskDone	Determine if an asynchronous task has completed.
FtpTaskResume	Resume execution of an asynchronous task.
FtpTaskSuspend	Suspend execution of an asynchronous task.
FtpTaskWait	Wait for an asynchronous task to complete.
FtpUninitialize	Terminate use of the library by the calling process.
FtpUploadFile	Upload a file from the local system to the server.

FtpValidateHostName	Validate the specified host name and return the resolved IP address.
FtpValidateUrl	Check the contents of a string to ensure it represents a valid URL.
FtpVerifyFile	Compare the contents of a local file against a file stored on the server.
FtpWaitForQueue	Wait for the transfer of all queued files to complete.
FtpWrite	Write data to the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAddQueuedFile Function

```
DWORD WINAPI FtpAddQueuedFile(
   HQUEUE hQueue,
   DWORD dwQueueMode,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nFileType,
   UINT nTimeout
);
```

The **FtpAddQueuedFile** function adds a new file to the queue with additional transfer options.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An integer value which specifies if the file should be downloaded from the server or uploaded to the server. This parameter must be one of the following options and cannot have a value of zero.

Constant	Description
FTP_QUEUE_DOWNLOAD	The file should be queued for downloading from the server to the local system.
FTP_QUEUE_UPLOAD	The file should be queued for uploading from the local system to the server.

lpszLocalFile

A null-terminated string which specifies the name of the local file to be queued for transfer. This parameter cannot be NULL or point to an empty string. If the file is being queued for an upload, the file must exist on the local system or the function will fail. If the file is being downloaded, it will be created or replaced on the local system.

lpszRemoteFile

A null-terminated string which specifies the URL to the location of the file on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator and can be any of the options used with the **FtpConnect** function. If this value is zero, the default options for the queue will be used. The most common options are:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the file transfer should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the

	server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the file transfer should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the file transfer should attempt to establish a secure connection with the server by default. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the file transfer should attempt to immediately establish secure TLS connection with the server by default. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the file transfer should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the file transfer should use the Secure Shell (SSH) protocol to establish the connection by default. This option will automatically be selected if the connection is established using port 22, the default port for SSH.

nFileType

An integer value which specifies the type of file being transferred. It can be one of the following values:

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **FtpRegisterFileType** function.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.

FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

nTimeout

The number of seconds to wait for a response from the server before failing the file transfer. If this value is zero, the default timeout period for the queue will be used.

Return Value

If the function succeeds, the return value is a unique identifier which corresponds to the queued file. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function will add a file to the current transfer queue. If successful, the value returned is a unique integer which identifies that file in the queue. To obtain information about the queued file, you can call the **FtpGetQueuedFile** function and provide this return value as the ***dwFileId*** parameter. To remove a file from the queue, use the **FtpRemoveQueuedFile** function.

Files can only be added to the queue when in an idle or paused state. If you attempt to add a file to the queue while the queue manager is transferring files, the function will fail. To determine the current state of the queue, call the **FtpGetQueueStatus** function

There are no fixed limits to the number of files which can be queued for transfer. To release the memory allocated for the queue, call the **FtpClearQueue** function. To determine the current status of the queue, including how many file transfers have been queued and how many are pending completion, call the **FtpGetQueueStatus** function.

If the local file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

This function will not immediately begin the process of uploading or downloading the file. It will only add the transfer request to the current queue. The file transfers are managed by a background worker thread which is created when the **FtpStartQueue** function is called.

The **FtpAddQueuedFileEx** function is an extended version of this function with additional parameters to specify a hostname, port number, user name and password. Because most file uploads require

authentication, using the extended function may be preferred to composing a URL which includes the account credentials.

Example

```
// Create a new queue
HQUEUE hQueue = FtpCreateQueue(INFINITE, FTP_TIMEOUT, FTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = FtpAddQueuedFile(
                     hQueue,
                     lpszLocalFile,
                     lpszRemoteFile,
                     FTP_OPTION_DEFAULT,
                     FILE_TYPE_AUTO,
                     FTP_TIMEOUT);

// Display information about the queued file
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;

    if (FtpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (FtpStartQueue(hQueue, FTP_QUEUE_ALL, 0, NULL, 0))
{
    FtpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
    FtpStopQueue(hQueue);
}

// Remove all files from the queue
FtpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFileEx, FtpCreateQueue, FtpDeleteQueue, FtpFindQueuedFile, FtpGetFirstQueuedFile, FtpGetNextQueuedFile, FtpGetQueuedFile, FtpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAddQueuedFileEx Function

```
DWORD WINAPI FtpAddQueuedFileEx(
   HQUEUE hQueue,
   DWORD dwQueueMode,
   LPCTSTR lpszHostName,
   UINT nHostPort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nFileType,
   UINT nTimeout,
   LPVOID lpvReserved
);
```

The **FtpAddQueuedFileEx** function adds a new file to the queue with additional transfer options.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An integer value which specifies if the file should be downloaded from the server or uploaded to the server. This parameter must be one of the following options and cannot have a value of zero.

Constant	Description
FTP_QUEUE_DOWNLOAD	The file should be queued for downloading from the server to the local system.
FTP_QUEUE_UPLOAD	The file should be queued for uploading from the local system to the server.

lpszHostName

The name of the server to establish a connection with. This may be a fully-qualified domain name or an IP address. If this parameter is NULL or points to an empty string, the ***lpszRemoteFile*** parameter must specify a complete URL which identifies the server as well as the path to the remote file.

nHostPort

The port number which should be used when establishing the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If this parameter is zero and a URL is specified with the ***lpszRemoteFile*** parameter, the port number included in the URL will be used as the default value.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the connection. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

A null-terminated string which specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

lpszLocalFile

A null-terminated string which specifies the name of the local file to be queued for transfer. This parameter cannot be NULL or point to an empty string. If the file is being queued for an upload, the file must exist on the local system or the function will fail. If the file is being downloaded, it will be created or replaced on the local system.

lpszRemoteFile

A null-terminated string which specifies the name of the file on the server. If the file is being downloaded, the file must exist on the server or the transfer will fail. If the file is being uploaded, the file will be created or replaced on the server. This parameter may specify a complete URL. Note that any values you specify as arguments to this function will override the values specified in the URL. If you want to use a URL as the remote file name, the ***lpszHostName*** parameter should be NULL and the ***nHostPort*** parameter should be zero.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator and can be any of the options used with the **FtpConnect** function. If this value is zero, the default options for the queue will be used. The most common options are:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the file transfer should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the file transfer should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the file transfer should attempt to establish a secure connection with the server by default. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the file transfer should attempt to immediately establish secure TLS connection with the server by default. This option is typically used when

	connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the file transfer should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the file transfer should use the Secure Shell (SSH) protocol to establish the connection by default. This option will automatically be selected if the connection is established using port 22, the default port for SSH.

nFileType

An integer value which specifies the type of file being transferred. It can be one of the following values:

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **FtpRegisterFileType** function.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

nTimeout

The number of seconds to wait for a response from the server before failing the file transfer. If this

value is zero, the default timeout period for the queue will be used.

lpvReserved

A reserved parameter which should always be NULL.

Return Value

If the function succeeds, the return value is a unique identifier which corresponds to the queued file. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function will add a file to the current transfer queue. If successful, the value returned is a unique integer which identifies that file in the queue. To obtain information about the queued file, you can call the **FtpGetQueuedFile** function and provide this return value as the ***dwFileId*** parameter. To remove a file from the queue, use the **FtpRemoveQueuedFile** function.

Files can only be added to the queue when in an idle or paused state. If you attempt to add a file to the queue while the queue manager is transferring files, the function will fail. To determine the current state of the queue, call the **FtpGetQueueStatus** function

There are no fixed limits to the number of files which can be queued for transfer. To release the memory allocated for the queue, call the **FtpClearQueue** function. To determine the current status of the queue, including how many file transfers have been queued and how many are pending completion, call the **FtpGetQueueStatus** function.

If the local file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

This function will not immediately begin the process of uploading or downloading the file. It will only add the transfer request to the current queue. The file transfers are managed by a background worker thread which is created when the **FtpStartQueue** function is called.

Example

```
// Create a new queue
HQUEUE hQueue = FtpCreateQueue(INFINITE, FTP_TIMEOUT, FTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = FtpAddQueuedFileEx(
                      hQueue,
                      FTP_QUEUE_DOWNLOAD,
                      lpszHostName,
                      FTP_PORT_DEFAULT,
                      lpszUserName,
                      lpszPassword,
                      lpszLocalFile,
                      lpszRemoteFile,
                      FTP_OPTION_DEFAULT,
                      FILE_TYPE_AUTO,
                      FTP_TIMEOUT,
                      NULL);

// Display information about the queued file
```

```
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;

    if (FtpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (FtpStartQueue(hQueue, FTP_QUEUE_ALL, 0, NULL, 0))
{
    FtpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
    FtpStopQueue(hQueue);
}

// Remove all files from the queue
FtpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpCreateQueue, FtpDeleteQueue, FtpFindQueuedFile, FtpGetFirstQueuedFile, FtpGetNextQueuedFile, FtpGetQueuedFile, FtpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAllocate Function

```
INT WINAPI FtpAllocate(
   HCLIENT hClient,
   DWORD dwFileLength,
   DWORD dwRecSize
);
```

The **FtpAllocate** function instructs the server to reserve sufficient storage to accommodate the new file being transferred.

Parameters

hClient

Handle to the client session.

dwFileLength

The number of bytes to allocate storage for on the server.

dwRecSize

The maximum record or page size for the file. A value of zero indicates that the file does not have a record or page structure, and the parameter is ignored.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function should be called immediately before the **FtpOpenFile** function.

This function is ignored by those servers which do not require that the maximum size of the file be declared beforehand. The most common FTP servers running under UNIX and Windows do not require that file space be pre-allocated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpOpenFile, FtpSetFileMode, FtpSetFileStructure, FtpSetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncConnect Function

```
HCLIENT WINAPI FtpAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **FtpAsyncConnect** function establishes a connection with the specified server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **FtpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If the secure port number is specified, an implicit TLS connection will be established by default.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description

FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be

	automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is

	established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members ***dwSize***, ***lpszCertStore*** and ***lpszCertName*** must be defined. Undefined structure members must be initialized to a value of zero or NULL and the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.

FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event occurs when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event occurs when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

To cancel asynchronous notification and return the client to a blocking mode, use the **FtpDisableEvents** function.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
// The Ipswitch WS_FTP server accepts the AUTH command to establish
// an explicit TLS session on the default FTP port
hClient = FtpAsyncConnect(lpszRemoteHost,
                          FTP_PORT_DEFAULT,
                          FTP_TIMEOUT,
                          FTP_OPTION_SECURE_EXPLICIT,
                          NULL,
                          hEventWnd,
                          uEventMsg);

// When the GlobalSCAPE Secure FTP server is configured in implicit
// authorization mode, it negotiates a secure session as soon as the
// connection is established and does not require a command
```

```
hClient = FtpAsyncConnect(lpszRemoteHost,
                          FTP_PORT_SECURE,
                          FTP_TIMEOUT,
                          FTP_OPTION_SECURE_IMPLICIT,
                          NULL,
                          hEventWnd,
                          uEventMsg);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncProxyConnect, FtpConnect, FtpCreateSecurityCredentials, FtpDeleteSecurityCredentials, FtpDisconnect, FtpGetSecurityInformation, FtpInitialize, FtpProxyConnect, FtpSetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncFileList Function

```
UINT WINAPI FtpAsyncFileList(
   HCLIENT hClient,
   LPCTSTR lpszDirectory,
   DWORD dwOptions,
   LPTSTR lpszBuffer,
   INT nMaxLength,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Return an unparsed list of files in the specified directory.

Parameters

hClient

Handle to the client session.

lpszDirectory

A null-terminated string that specifies the name of a directory and/or a wildcard file mask. The format of the directory name must match the file naming conventions of the server. If this parameter is NULL or points to an empty string, the current working directory will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_LIST_DEFAULT	This option specifies the server should return a complete listing of files in the specified directory with as much detail as possible. This typically means that the file size, date, ownership and access rights will be returned to the client. Information about the files are returned in lines of text, with each line terminated by carriage return and linefeed (CRLF) characters. The exact format of the data returned is specific to the server operating system.
FTP_LIST_NAMEONLY	This option specifies the server should only return a list of file names, with no additional information about the file. Each file name is terminated by carriage return and linefeed (CRLF) characters.

lpszBuffer

A null-terminated string buffer that will contain the list of files when the function returns. This buffer should be large enough to store the complete file listing and a terminating null character. If the buffer is smaller than the total amount of data returned by the server, the data will be truncated. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more

information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpAsyncFileList** function returns a list of files in the specified directory, copying the data to a string buffer. This function is similar to the **FtpGetFileList** function, however it retrieves the list of files asynchronously using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file listing is being returned by the server.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, such as with the example listed below, then you must ensure that your code does not return from the function while the directory listing is being retrieved. In the example, this is achieved by calling the **FtpTaskWait** function. You can also perform other operation and poll the status of the task by calling the **FtpTaskDone** function. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpszBuffer* parameter on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the directory listing has completed. This enables your application to know when the directory listing is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

This function can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the library. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the FTP_LIST_NAMEONLY option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This function is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

Example

```
TCHAR szFileList[MAXFILELISTSIZE];
UINT nTaskId;

nTaskId = FtpAsyncFileList(hClient,
```

```
                            NULL,
                            FTP_LIST_DEFAULT,
                            szFileList,
                            MAXFILELISTSIZE,
                            NULL, 0);

    if (nTaskId == 0)
    {
        _tprintf(_T("Unable to list files (error 0x%08lx)\n"), FtpGetLastError());
        return;
    }
    else
    {
        DWORD dwError = NO_ERROR;
        DWORD dwElapsed = 0;

        FtpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);
        _tprintf(_T("Asynchronous file listing returned in %lu milliseconds\n"),
    dwElapsed);
    }
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, FtpGetFileList, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncGetData Function

```
UINT WINAPI FtpAsyncGetData(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Copies the contents of a file on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the ***lpEventProc*** parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncGetData** function is used to download the contents of a remote file into a local buffer.

This function is similar to the **FtpGetData** function, however it retrieves the contents of the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being copied to the local system.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, such as with the example listed below, then you must ensure that your code does not return from the function while the data is being downloaded. In the example, this is achieved by calling the **FtpTaskWait** function. You can also perform other operation and poll the status of the task by calling the **FtpTaskDone** function. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the ***lpvBuffer*** and ***lpdwLength*** parameters on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

The ***lpBuffer*** parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the file or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the ***lpdwLength*** parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then ***lpdwLength*** must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
UINT nTaskId;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter

nTaskId = FtpAsyncGetData(hClient,
                          lpszFileName,
                          &hgblBuffer,
                          &dwLength,
                          0, NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    FtpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);
```

```
    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncGetFile, FtpAsyncPutData, FtpAsyncPutFile, FtpEventProc, FtpGetFile, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncGetFile Function

```
UINT WINAPI FtpAsyncGetFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Downloads the specified file from the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncGetFile** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **FtpGetFile** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the FTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Example

```
UINT nTaskId;

// Begin a file transfer in the background

nTaskId = FtpAsyncGetFile(hClient,
                          lpszLocalFile,
                          lpszRemoteFile,
                          FTP_TRANSFER_DEFAULT,
                          0,
                          NULL,
                          0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;
```

```
    // Wait for the transfer to complete
    FtpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncGetData, FtpAsyncPutData, FtpAsyncPutFile, FtpEventProc, FtpGetFile, FtpTaskDone, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncGetFileEx Function

```
UINT WINAPI FtpAsyncGetFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the

lpEventProc parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncGetFileEx** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **FtpGetFileEx** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the FTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncPutFileEx, FtpEventProc, FtpGetFileEx, FtpPutFileEx, FtpTaskDone, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncProxyConnect Function

```
HCLIENT WINAPI FtpAsyncProxyConnect(
   UINT nProxyType,
   LPCTSTR lpszProxyHost,
   UINT nProxyPort,
   LPCTSTR lpszProxyUser,
   LPCTSTR lpszProxyPassword,
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **FtpAsyncProxyConnect** function establishes a connection through a proxy server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **FtpProxyConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

nProxyType

An identifier which specifies the type of proxy server that is being connected to. This value must be defined as one of the following values:

Value	Description
FTP_PROXY_NONE	This value specifies that no proxy server is being used. In this case, the **FtpConnect** function is called directly, ignoring the proxy parameters.
FTP_PROXY_USER	This value specifies that the client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
FTP_PROXY_LOGIN	This value specifies that the client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
FTP_PROXY_OPEN	This value specifies that the client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
FTP_PROXY_SITE	This value specifies that the client is logged into the server. The SITE command is sent, specifying the host name, followed by the

	username and the password.
FTP_PROXY_OTHER	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the FtpCommand function to send specific command sequences.

lpszProxyHost

The name of the proxy server to connect through; this may be a fully-qualified domain name or an IP address.

nProxyPort

The port number the proxy server is listening on; a value of zero specifies that the default port number should be used.

lpszProxyUser

The user name used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a username is required.

lpszProxyPassword

The password used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a password is required.

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the

	server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur

	when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of

	IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members ***dwSize***, ***lpszCertStore*** and ***lpszCertName*** must be defined. Undefined structure members must be initialized to a value of zero or NULL and the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

The username and password that is used to authenticate the client with the proxy server are not the same as those used to login to the target server. Once a connection has been established with the proxy server, the client must call the **FtpLogin** function to actually login to the server and begin a file

transfer.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event occurs when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event occurs when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

To cancel asynchronous notification and return the client to a blocking mode, use the

FtpDisableEvents function.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncConnect, FtpConnect, FtpCreateSecurityCredentials, FtpDeleteSecurityCredentials, FtpDisconnect, FtpGetSecurityInformation, FtpInitialize, FtpLogin, FtpProxyConnect, FtpSetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncPutData Function

```
UINT WINAPI FtpAsyncPutData(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPVOID lpvBuffer,
   DWORD dwLength,
   DWORD dwReserved,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Copies the contents of the specified buffer to a file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpvBuffer

The data that will be copied to the server and stored in the specified file.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncPutData** function is used to upload the contents of a local buffer to the server. This function is similar to the **FtpPutData** function, however it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent to the server.

Because this function works asynchronously, it is important that the memory allocated for the buffer is

not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, ensure that your code does not return from the function while the data is being uploaded. This can be achieved by calling the **FtpTaskWait** function or periodically calling the **FtpTaskDone** function to determine if the transfer has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the ***lpvBuffer*** parameter on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

If the ***lpvBuffer*** parameter is pointing to a Unicode string, it is important to note that the value of the ***dwLength*** parameter should specify the number of bytes, not the number of characters. When using UTF-16, each character is two bytes long and therefore the length of the buffer is effectively double the length of the string. Because Unicode strings can contain null characters, you must also set the current file type to FILE_TYPE_IMAGE prior to calling this function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncGetData, FtpAsyncGetFile, FtpAsyncPutFile, FtpEventProc, FtpPutData, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncPutFile Function

```
UINT WINAPI FtpAsyncPutFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Uploads the specified file from the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncPutFile** function is used to upload the contents of a local file to the server. This function is similar to the **FtpPutFile** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the FTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncGetData, FtpAsyncGetFile, FtpAsyncPutData, FtpEventProc, FtpPutFile, FtpTaskDone, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAsyncPutFileEx Function

```
UINT WINAPI FtpAsyncPutFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Uploads the specified file from the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the

lpEventProc parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncPutFileEx** function is used to upload the contents of a local file to the server. This function is similar to the **FtpPutFileEx** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the FTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAsyncGetFileEx, FtpEventProc, FtpGetFileEx, FtpPutFileEx, FtpTaskDone, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAttachThread Function

```
DWORD WINAPI FtpAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **FtpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **FtpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **FtpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **FtpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **FtpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpDisconnect, FtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCancel Function

```
INT WINAPI FtpCancel(
   HCLIENT hClient
);
```

The **FtpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

When the **FtpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This function may be called during a blocking file transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCancelQueuedFile Function

```
BOOL WINAPI FtpCancelQueuedFile(
    HQUEUE hQueue,
    DWORD dwFileId
);
```

The **FtpCancelQueuedFile** function cancels the queued file transfer.

Parameters

hQueue

A handle to a file transfer queue.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. If this value is zero, the function will cancel the current file transfer in progress. If queue transfers have been suspended, this parameter cannot be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

When this function is called, the queued file transfer may not immediately stop. An internal flag is set which causes the file transfer to exit with an error and the queue manager will begin processing the next file in the queue. If the queue is in an idle state, this function will fail.

It is permitted to call **FtpCancelQueuedFile** from within a queue event handler to cancel the current file transfer. In this case, the *dwFileId* parameter should be zero. If you specify a file which has not been transferred, it will be flagged as canceled and skipped by the queue manager when processing the queue. If you specify a file which has already been processed, this function will fail.

A canceled file transfer is also considered a failed transfer. When you call **FtpGetQueueStatus** after a queued file transfer is canceled, the *dwFailedFiles* member of the FTPQUEUESTATUS structure will be incremented to reflect this change. You can determine the status of an individual file transfer by calling the **FtpGetQueuedFile** function and checking the value of the *dwQueueFlags* member of the FTPQUEUEDFILE structure.

The **FtpResetQueue** function can be used to reset the state of previously canceled transfers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancel, FtpGetQueuedFile, FtpGetQueueStatus, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpSuspendQueue, FtpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpChangeDirectory Function

```
INT WINAPI FtpChangeDirectory(
   HCLIENT hClient,
   LPCTSTR lpszDirectory
);
```

The **FtpChangeDirectory** function changes the current working directory for the client session.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the CWD command to change the current working directory. The user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectoryUp, FtpCloseDirectory, FtpCommand, FtpGetDirectory, FtpGetFirstFile, FtpGetNextFile, FtpGetResultCode, FtpGetResultString, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpChangeDirectoryUp Function

```
INT WINAPI FtpChangeDirectoryUp(
    HCLIENT hClient
);
```

The **FtpChangeDirectoryUp** function changes directory to the parent of the current working directory.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the CDUP command to the server. This command is a special case of the CWD command, and is included to simplify transferring between directory trees on those operating systems which have different syntaxes for naming the parent directory. The current user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpChangeDirectory, FtpCommand, FtpGetDirectory, FtpGetResultCode, FtpGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpClearQueue Function

```
BOOL WINAPI FtpClearQueue(
    HQUEUE hQueue
);
```

The **FtpClearQueue** function removes all files from the queue.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function can only be called when the queue is in an idle state. An error will be returned if the function is called while the queue manager is paused or actively transferring files in the queue. To determine the current state of the queue, call the **FtpGetQueueStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancelQueuedFile, FtpGetQueueStatus, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpStopQueue, FtpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCloseDirectory Function

```
INT WINAPI FtpCloseDirectory(
    HCLIENT hClient
);
```

The **FtpCloseDirectory** function closes the data socket connection to the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function must be called after all of the file information from the server has been returned. Because directory information is returned on the data channel, no file transfers can take place while a directory is being read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetDirectoryFormat, FtpGetFileStatus, FtpGetFirstFile, FtpGetNextFile, FtpOpenDirectory, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCloseFile Function

```
INT WINAPI FtpCloseFile(
   HCLIENT hClient
);
```

The **FtpCloseFile** function flushes the internal client buffers and closes the data socket connection to the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the socket is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the function returns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpOpenFile, FtpRead, FtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCommand Function

```
INT WINAPI FtpCommand(
   HCLIENT hClient,
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **FtpCommand** function sends a command to the server and returns the result code back to the caller.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the function succeeds, the return value is the result code returned by the server. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function should only be used when the application needs to send a custom, site-specific command or send a command that is not directly supported by this API. This function should never be used to issue a command that opens a data channel. If the application needs to transform data as it is being sent or received, and cannot use the **FtpGetFile** or **FtpPutFile** functions, then use the **FtpOpenFile** function to open a data channel with the server.

By default, file names which are sent to the server using the **FtpCommand** function are sent as ANSI characters. If the Unicode version of the function is used, the file name will be converted from Unicode to ANSI using the current codepage. If the server supports UTF-8 encoded file names, the **FtpSetFileNameEncoding** function can be used to specify that file names with non-ASCII characters should be sent as UTF-8 encoded values. It is important to note that this option is only available if the server advertises support for UTF-8 and permits that encoding type.

Because the **FtpCommand** function communicates directly with the server using the control connection, the idle timeout period configured with **FtpSetIdleTimeout** applies to this function as well. If a session remains inactive for longer than the configured timeout, the server may close the connection before the command is sent or before a response is received. Adjusting the idle timeout can help prevent unexpected disconnects during periods of infrequent command activity, especially in interactive or long-running operations.

This function is not supported when using SFTP because that protocol does not provide a mechanism for sending arbitrary command strings. Unlike the FTP protocol, which uses a command-response model over a control channel, SFTP uses a structured binary protocol without support for custom commands. If this function is called while connected to an SFTP server, it will fail and return an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFile, FtpGetFileNameEncoding, FtpGetIdleTimeout, FtpGetResultCode, FtpGetResultString, FtpOpenFile, FtpPutFile, FtpSetFileNameEncoding, FtpSetIdleTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpConnect Function

```
HCLIENT WINAPI FtpConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **FtpConnect** function establishes a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the

	server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur

	when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP

	commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members ***dwSize***, ***lpszCertStore*** and ***lpszCertName*** must be defined. Undefined structure members must be initialized to a value of zero or NULL and the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **FtpConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
// Connect to a server using a standard (non-secure) connection on
// the default port. The username and password are not encrypted.

hClient = FtpConnect(lpszRemoteHost,
                     FTP_PORT_DEFAULT,
                     lpszUserName,
                     lpszPassword,
                     FTP_TIMEOUT,
                     FTP_OPTION_DEFAULT,
                     NULL);

// Connect to a server using the default port and then initiate a
// secure connection using the AUTH TLS command

hClient = FtpConnect(lpszRemoteHost,
                     FTP_PORT_DEFAULT,
                     lpszUserName,
                     lpszPassword,
                     FTP_TIMEOUT,
                     FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_EXPLICIT,
                     NULL);

// Connect to a server on port 990 and immediately initiate a
// secure connection as soon as the connection is established

hClient = FtpConnect(lpszRemoteHost,
                     FTP_PORT_SECURE,
```

```
                        lpszUserName,
                        lpszPassword,
                        FTP_TIMEOUT,
                        FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_IMPLICIT,
                        NULL);

// Connect to a server on port 22 using the Secure Shell (SFTP)
// protocol to transfer files

hClient = FtpConnect(lpszRemoteHost,
                        FTP_PORT_SSH,
                        lpszUserName,
                        lpszPassword,
                        FTP_TIMEOUT,
                        FTP_OPTION_SECURE_SHELL,
                        NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCreateSecurityCredentials, FtpDeleteSecurityCredentials, FtpDisconnect, FtpGetSecurityInformation, FtpInitialize, FtpProxyConnect, FtpSetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpConnectUrl Function

```
HCLIENT WINAPI FtpConnectUrl(
    LPCTSTR lpszUrl,
    UINT nTimeout,
    DWORD dwOptions
);
```

The **FtpConnectUrl** function establishes a connection with the specified server using a URL.

Parameters

lpszUrl

A null-terminated string which specifies the URL for the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require

	authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to

	a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is

	synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpConnectUrl** function is a high-level function that uses an FTP URL to establish a connection with a server. Unlike the other connection related functions such as **FtpConnect**, this function does more than simply connect to the server. It will also authenticate the client session, change the current working directory and set the default file transfer mode. By default, this function will always place the client in passive mode, ensuring the broadest compatibility with most servers.

The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. The paths in an FTP URL are relative to the home directory of the user account and are not absolute paths starting at the root directory on the server. If a file name is also specified in the URL, it will be ignored and only the file path will be used. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The **FtpConnectUrl** function is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection functions. If you only need to upload or download a file using a URL, then refer to the **FtpUploadFile** and **FtpDownloadFile** functions.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **FtpConnectUrl** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when

a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
HCLIENT hClient;
LPCTSTR lpszUrl = _T("ftp://ftp.sockettools.com/");

// Connect to the site specified by the URL
hClient = FtpConnectUrl(lpszUrl, FTP_TIMEOUT, FTP_OPTION_DEFAULT);

if (hClient == INVALID_CLIENT)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpDisconnect, FtpDownloadFile, FtpUploadFile, FtpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCreateDirectory Function

```
INT WINAPI FtpCreateDirectory(
   HCLIENT hClient,
   LPCTSTR lpszDirectory
);
```

The **FtpCreateDirectory** function creates the specified directory on the server.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the MKD command to create the directory. The user must have the appropriate permission to create the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpGetDirectory, FtpRemoveDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCreateQueue Function

```
HQUEUE WINAPI FtpCreateQueue(
    UINT nMaxFiles,
    UINT nTimeout,
    DWORD dwOptions
);
```

The **FtpCreateQueue** function creates new file transfer queue.

Parameters

nMaxFiles

An unsigned integer which specifies the maximum number of files which can be added to the queue. The constant INFINITE can be used to specify that there is no fixed limit to the number of files in the queue.

nTimeout

An unsigned integer which specifies the default timeout for all queued file transfers. If this value is zero, a reasonable default timeout period will be used. This timeout period is used when a file is added to the queue without providing a timeout period for that specific file transfer. If a timeout period is specified for a particular file, it will override this value.

dwOptions

An unsigned integer value which specifies one or more default options for all queued files. This parameter is constructed by using a bitwise operator and can be any of the options used with the **FtpConnect** function. If transfer options are specified for a particular file, it will override this value. The most common options are:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for queued file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies queued transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies queued transfers should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an

	unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies queued file transfers should attempt to establish a secure connection with the server by default. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies queued file transfers should attempt to immediately establish secure TLS connection with the server by default. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies queued file transfers should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies queued file transfers should use the Secure Shell (SSH) protocol to establish the connection by default. This option will automatically be selected if the connection is established using port 22, the default port for SSH.

Return Value

If the function succeeds, the return value is a handle to the queue which can be used with subsequent function calls. If the function fails, the return value is INVALID_QUEUE. To get extended error information, call **FtpGetLastError**.

Remarks

If the *nMaxFiles* parameter is INFINITE, memory will be dynamically allocated on the process heap with no limit to the number of files which can be queued. If there is a logic error which causes the application to recursively add files to the queue, or repeatedly queue the same file, this can result in virtual memory being exhausted for the process. You can avoid this by specifying a reasonable maximum queue size, which will cause the **FtpAddQueuedFile** function to fail if that limit is exceeded.

The memory allocated for the queue will be released when the **FtpDeleteQueue** function is called.

Example

```
// Create a new queue
HQUEUE hQueue = FtpCreateQueue(INFINITE, FTP_TIMEOUT, FTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = FtpAddQueuedFile(
                    hQueue,
                    lpszLocalFile,
```

```
                    lpszRemoteFile,
                    FTP_OPTION_DEFAULT,
                    FILE_TYPE_AUTO,
                    FTP_TIMEOUT);

// Display information about the queued file
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;

    if (FtpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (FtpStartQueue(hQueue, FTP_QUEUE_ALL, 0, NULL, 0))
{
    FtpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
    FtpStopQueue(hQueue);
}

// Remove all files from the queue
FtpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpAddQueuedFile, FtpDeleteQueue, FtpClearQueue, FtpStartQueue, FtpStopQueue, FtpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCreateSecurityCredentials Function

```
BOOL WINAPI FtpCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **FtpCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

A value which specifies one or more options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **FtpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **FtpConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you

can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPCTSTR lpszCertStore = _T("HKCU:MY");   // The current user's personal certificate
store
LPCTSTR lpszCertName = _T("ClientCert"); // The client certificate name
LPSECURITYCREDENTIALS lpSecCred = NULL;

FtpCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                             CREDENTIAL_STORE_CURRENT_USER,
                             NULL,
                             NULL,
                             lpszCertStore,
                             lpszCertName,
                             NULL,
                             &lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpDeleteSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDeleteFile Function

```
INT WINAPI FtpDeleteFile(
   HCLIENT hClient,
   LPCTSTR lpszFileName
);
```

The **FtpDeleteFile** function deletes the specified file from the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The current user must have the appropriate permission to delete the file, or an error will be returned by the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFile, FtpPutFile, FtpRenameFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDeleteQueue Function

```
BOOL WINAPI FtpDeleteQueue(
    HQUEUE hQueue
);
```

The **FtpDeleteQueue** function deletes the specified file transfer queue, releasing all memory allocated for the queued files.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function can only be called when the queue is in an idle state. If the queue manager is in the process of uploading or downloading files, the function will fail and return a value of zero. You can determine the current state of the queue by calling the **FtpGetQueueStatus** function. It is not necessary to explicitly clear the queue prior to calling this function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCreateQueue, FtpGetQueueStatus, FtpCancelQueuedFile, FtpClearQueue, FtpStartQueue, FtpStopQueue, FtpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDeleteSecurityCredentials Function

```
VOID WINAPI FtpDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **FtpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)
    FtpDeleteSecurityCredentials(&lpSecCred);

FtpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCreateSecurityCredentials, FtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDisableEvents Function

```
INT WINAPI FtpDisableEvents(
    HCLIENT hClient
);
```

The **FtpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpDisableEvents** function is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this function has been called.

This function is automatically called if the client has event notification enabled, and the **FtpDisconnect** function is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpAsyncConnect, FtpAsyncProxyConnect, FtpEnableEvents, FtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDisableTrace Function

```
BOOL WINAPI FtpDisableTrace();
```

The **FtpDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, a value of zero is returned.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDisconnect Function

```
INT WINAPI FtpDisconnect(
   HCLIENT hClient
);
```

The **FtpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpProxyConnect, FtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDownloadFile Function

```
BOOL WINAPI FtpDownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPFTPTRANSFERSTATUS lpStatus
   FTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **FtpDownloadFile** function downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the complete URL of the file to be downloaded to the local system. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.

FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpDownloadFile** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then download the file to the local system. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be downloaded as a text or binary file. A value of "a" specifies that the file should be downloaded as a text file. A value of "i" specifies that the file should be downloaded as a binary file. If the type is not explicitly specified, the file will be downloaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpDownloadFile** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpGetFile** to download the file.

Example

```
FTPTRANSFERSTATUS ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile = _T("ftp://ftp.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = FtpDownloadFile(lpszLocalFile,
                          lpszRemoteFile,
                          FTP_OPTION_PASSIVE,
                          &ftpStatus,
                          NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, FtpGetFile, FtpGetFileZoneInfo, FtpGetTransferStatus, FtpSetFileZoneInfo, FtpUploadFile, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDownloadFileEx Function

```
BOOL WINAPI FtpDownloadFileEx(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPFTPTRANSFERSTATUSEX lpStatus
   FTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **FtpDownloadFileEx** function downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an

	unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpDownloadFileEx** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then download the file to the local system. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be downloaded as a text or binary file. A value of "a" specifies that the file should be downloaded as a text file. A value of "i" specifies that the file should be downloaded as a binary file. If the type is not explicitly specified, the file will be downloaded as a binary file.

The ***lpStatus*** parameter can be used by the application to determine the final status of the transfer,

including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatusEx** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpDownloadFileEx** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpGetFileEx** to download the file.

Example

```
FTPTRANSFERSTATUSEX ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile = _T("ftp://ftp.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = FtpDownloadFileEx(lpszLocalFile,
                            lpszRemoteFile,
                            FTP_OPTION_PASSIVE,
                            &ftpStatus,
                            NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, FtpGetFileEx, FtpGetTransferStatusEx, FtpUploadFileEx, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnableEvents Function

```
INT WINAPI FtpEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **FtpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **FtpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpEnableEvents** function is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.

FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event occurs when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event occurs when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **FtpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpDisableEvents, FtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnableFeature Function

```
BOOL WINAPI FtpEnableFeature(
    HCLIENT hClient,
    DWORD dwFeature,
    BOOL bEnable
);
```

The **FtpEnableFeature** function enables or disables a specific server feature available to the client.

Parameters

hClient

Handle to the client session.

dwFeature

An unsigned integer which specifies the feature to be enabled to disabled for the current client session. Refer to the documentation for the **FtpGetFeatures** function for a list of available features.

bEnable

A boolean flag which specifies if the feature should be enabled or disabled. If the value is non-zero, the library will attempt to use that feature on the server. If the value is zero, the feature is disabled. If an application calls a function which requires a specific feature and that feature is disabled, the function will fail with an error indicating the feature is not supported.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return a value of zero. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpEnableFeature** function is used to enable or disable a specific feature for the current session. When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. An application can use the **FtpEnableFeature** function to control what commands will be sent to the server, or re-enable a command that was previously disabled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetFeatures, FtpSetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnableTrace Function

```
BOOL WINAPI FtpEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **FtpEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEndOfFile Function

```
BOOL WINAPI FtpEndOfFile(
   HCLIENT hClient
);
```

The **FtpEndOfFile** function is used to determine if the end-of-file has been reached while reading the contents of a remote file.

Parameters

hClient

Handle to the client session.

Return Value

If the end-of-file has been reached, the function returns a non-zero value. If the client handle is invalid, or the end-of-file has not been reached, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCloseFile, FtpOpenFile, FtpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnumFiles Function

```
INT WINAPI FtpEnumFiles(
   HCLIENT hClient,
   LPCTSTR lpszDirectory,
   LPCTSTR lpszFileMask,
   DWORD dwOptions,
   LPFTPFILESTATUS lpFileList,
   INT nMaxFiles
);
```

The **FtpEnumFilesEx** function populates an array of structures that contain information about the files in a directory.

Parameters

hClient

Handle to the client session.

lpszDirectory

A null-terminated string that specifies the name of a directory on the server. If this parameter is NULL or points to an empty string, the function will return the files in the current working directory. This string cannot contain wildcard characters and must specify a valid directory name that exists on the server.

lpszFileMask

A null-terminated string that specifies a wildcard file mask that is used to return a subset of files in the directory. If this parameter is NULL or an empty string then all of the files in the directory will be returned.

dwOptions

An unsigned integer value that specifies one or more options. This parameter can be a combination of one or more of the following values:

Value	Description
FTP_ENUM_DEFAULT	The function will return both regular files and subdirectories.
FTP_ENUM_FILE	The function will return only regular files.
FTP_ENUM_DIRECTORY	The function will return only subdirectories.
FTP_ENUM_FULLPATH	The function will return the full path of the file or subdirectory.

lpFileList

A pointer to an array of FTPFILESTATUS structures which contains information about each of the files in the specified directory. This parameter cannot be NULL, and the array must be large enough to store the number of files specified by the ***nMaxFiles*** parameter.

nMaxFiles

An integer value that specifies the maximum number of files that should be returned. This value must be greater than zero and the ***lpFileList*** parameter must provide an array that is large enough to store information about each file.

Return Value

If the function succeeds, the return value is the number of files returned by the function. If the

directory is empty or there are no files that match the specified wildcard file mask, the function will return zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpEnumFiles** function provides a high-level interface for obtaining a list of available files in a directory on the server in a single function call. This is an alternative to opening a directory and returning information about each file by calling the **FtpGetNextFile** function in a loop.

This function temporarily changes the current working directory to the directory specified by the *lpszDirectory* parameter. The current working directory will be restored to its original value when the function returns. The user must have the appropriate permissions to access the directory or this function will fail.

To obtain information on a subset of files in the directory, you can specify a wildcard file mask. For FTP and FTPS (TLS) sessions, this value is passed as a parameter to the LIST command and the server performs the wildcard matching. For SFTP (SSH) sessions the wildcard matching is performed by the library, and the standard conventions for Windows file wildcards are used.

By default, the *szFileName* member for each **FTPFILESTATUS** structure will contain the base file name. If the FTP_ENUM_FULLPATH option is specified, the function will return the full path name to the file. The library must be able to automatically determine the path delimiter that is used by the server. This is done by examining how the server identifies itself, the current directory format and the path the server returns for the current working directory. For example, UNIX based servers use the forward slash as a path delimiter. If the function cannot determine what the appropriate path delimiter is, it will ignore this option and return only the base file name.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled. If the directory contains files that are larger than 4GB, the **FtpEnumFilesEx** function should be used to obtain the correct file size.

Example

```
LPFTPFILESTATUS lpFileList = new FTPFILESTATUS[MAXFILECOUNT];

// Return all of the regular files in the current working directory
INT nResult = FtpEnumFiles(hClient, NULL, NULL, FTP_ENUM_FILE,
                           lpFileList, MAXFILECOUNT);

if (nResult == FTP_ERROR)
{
    DWORD dwError = FtpGetLastError();
    _tprintf(_T("FtpEnumFiles failed, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("FtpEnumFiles returned %d files\n"), nResult);
for (INT nIndex = 0; nIndex < nResult; nIndex++)
{
    _tprintf(_T("file=\"%s\" size=%lu\n"), lpFileList[nIndex].szFileName,
                                           lpFileList[nIndex].dwFileSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEnumFilesEx, FtpGetFileStatus, FtpGetFirstFile, FtpGetNextFile, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnumFilesEx Function

```
INT WINAPI FtpEnumFilesEx(
   HCLIENT hClient,
   LPCTSTR lpszDirectory,
   LPCTSTR lpszFileMask,
   DWORD dwOptions,
   LPFTPFILESTATUSEX lpFileList,
   INT nMaxFiles
);
```

The **FtpEnumFilesEx** function populates an array of structures that contain information about the files in a directory. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszDirectory

A null-terminated string that specifies the name of a directory on the server. If this parameter is NULL or points to an empty string, the function will return the files in the current working directory. This string cannot contain wildcard characters and must specify a valid directory name that exists on the server.

lpszFileMask

A null-terminated string that specifies a wildcard file mask that is used to return a subset of files in the directory. If this parameter is NULL or an empty string then all of the files in the directory will be returned.

dwOptions

An unsigned integer value that specifies one or more options. This parameter can be a combination of one or more of the following values:

Value	Description
FTP_ENUM_DEFAULT	The function will return both regular files and subdirectories.
FTP_ENUM_FILE	The function will return only regular files.
FTP_ENUM_DIRECTORY	The function will return only subdirectories.
FTP_ENUM_FULLPATH	The function will return the full path of the file or subdirectory.

lpFileList

A pointer to an array of FTPFILESTATUSEX structures which contains information about each of the files in the specified directory. This parameter cannot be NULL, and the array must be large enough to store the number of files specified by the ***nMaxFiles*** parameter.

nMaxFiles

An integer value that specifies the maximum number of files that should be returned. This value must be greater than zero and the ***lpFileList*** parameter must provide an array that is large enough to store information about each file.

Return Value

If the function succeeds, the return value is the number of files returned by the function. If the

directory is empty or there are no files that match the specified wildcard file mask, the function will return zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpEnumFilesEx** function provides a high-level interface for obtaining a list of available files in a directory on the server in a single function call. This is an alternative to opening a directory and returning information about each file by calling the **FtpGetNextFileEx** function in a loop.

This function temporarily changes the current working directory to the directory specified by the *lpszDirectory* parameter. The current working directory will be restored to its original value when the function returns. The user must have the appropriate permissions to access the directory or this function will fail.

To obtain information on a subset of files in the directory, you can specify a wildcard file mask. For FTP and FTPS (TLS) sessions, this value is passed as a parameter to the LIST command and the server performs the wildcard matching. For SFTP (SSH) sessions the wildcard matching is performed by the library, and the standard conventions for Windows file wildcards are used.

By default, the *szFileName* member for each **FTPFILESTATUSEX** structure will contain the base file name. If the FTP_ENUM_FULLPATH option is specified, the function will return the full path name to the file. The library must be able to automatically determine the path delimiter that is used by the server. This is done by examining how the server identifies itself, the current directory format and the path the server returns for the current working directory. For example, UNIX based servers use the forward slash as a path delimiter. If the function cannot determine what the appropriate path delimiter is, it will ignore this option and return only the base file name.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
LPFTPFILESTATUSEX lpFileList = new FTPFILESTATUSEX[MAXFILECOUNT];

// Return all of the regular files in the current working directory
INT nResult = FtpEnumFilesEx(hClient, NULL, NULL, FTP_ENUM_FILE,
                             lpFileList, MAXFILECOUNT);

if (nResult == FTP_ERROR)
{
    DWORD dwError = FtpGetLastError();
    _tprintf(_T("FtpEnumFilesEx failed, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("FtpEnumFilesEx returned %d files\n"), nResult);
for (INT nIndex = 0; nIndex < nResult; nIndex++)
{
    _tprintf(_T("file=\"%s\" size=%I64d\n"), lpFileList[nIndex].szFileName,
                                             lpFileList[nIndex].uiFileSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEnumFiles, FtpGetFileStatusEx, FtpGetFirstFileEx, FtpGetNextFileEx, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnumQueuedFiles Function

```
LONG WINAPI FtpEnumQueuedFiles(
   HQUEUE hQueue,
   LPCTSTR lpszFileMask,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPDWORD lpFileList,
   LONG nMaxFiles
);
```

The **FtpEnumQueuedFiles** function returns a list of files in the current transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

lpszFileMask

A null-terminated string which specifies a file name which can include wildcards. Only those files which match this value will be enumerated. The character '?' will match against any single character, and '*' will match any number of characters. If this parameter is NULL or points to an empty string, all eligible files in the queue will be matched.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. Only those files which match the queue mode will be enumerated. This is a bitmask which may be one or more of the following values:

Constant	Description
FTP_QUEUE_ALL	Match the file name to all files in the queue.
FTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
FTP_QUEUE_UPLOAD	Match file names which are queued for upload.
FTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the enumeration. If this value is zero, no files will be excluded; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_QUEUE_FLAG_NONE	No files should be excluded.
FTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
FTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.

FTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **FtpCancelQueuedFile** function has been called and a queued file is in the process of being uploaded or downloaded.

lpFileList

A pointer to an array of unsigned integer values which will contain the unique file identifiers for each matching file in the queue. This parameter must specify an array large enough to store all of the file identifiers, otherwise the function will fail with a ST_ERROR_BUFFER_TOO_SMALL error. If this parameter is NULL, the function will return the number of matching files.

nMaxFiles

An integer value which specifies the maximum number of file identifiers which can be copied into the *lpFileList* array. If the *lpFileList* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of matching files. If the queue is empty or there are no matching files, this function will return zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function populates an array of unique file identifiers which can be used to obtain information about the files in the current queue. The values returned in the *lpFileList* array can be used in conjunction with the **FtpGetQueuedFile** function. This provides an alternative to using the **FtpGetFirstQueuedFile** and **FtpGetNextQueuedFile** functions.

The application should not make any assumptions about the value of the file identifiers returned by this function. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

Example

```
// Get the total number of files in the queue
LONG nFiles = FtpEnumQueuedFiles(hQueue, NULL, FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE,
NULL, 0)

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];

    nFiles = FtpEnumQueuedFiles(hQueue, NULL, FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE,
pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
        FTPQUEUEDFILE queuedFile;

        if (FtpGetQueuedFile(hQueue, pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
```

```
                _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                 break;

        case FTP_QUEUE_UPLOAD:
                _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                 break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpFindQueuedFile, FtpGetFirstQueuedFile, FtpGetNextQueuedFile, FtpGetQueuedFile, FtpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnumTasks Function

```
INT WINAPI FtpEnumTasks(
   UINT *lpTasks,
   INT nMaxTasks,
   DWORD dwOptions
);
```

Return a list of active, suspended or finished asynchronous tasks.

Parameters

lpTasks

A pointer to an array of unsigned integer values that will contain unique task identifiers when the function returns. If this parameter is NULL, the function will return the number of tasks.

nMaxTasks

An integer value that specifies the maximum number of task identifiers that may be copied into the ***lpTasks*** array. If the ***lpTasks*** parameter is NULL, this value must be zero.

dwOptions

An unsigned integer that specifies the type of asynchronous tasks that may be returned by this function. It may be a combination of the following values:

Value	Description
FTP_TASK_DEFAULT	The list of asynchronous task IDs should include both active and suspended tasks. This option is the same as specifying both the FTP_TASK_ACTIVE and FTP_TASK_SUSPENDED options.
FTP_TASK_ACTIVE	The list of asynchronous task IDs should include those tasks which are currently active. An active task represents a background connection to a server that is in the process of performing the requested action, such as uploading or downloading a file.
FTP_TASK_SUSPENDED	The list of asynchronous task IDs should include those tasks which have been suspended. A suspended task represents a background connection that has been established, but the worker thread is not scheduled for execution.
FTP_TASK_FINISHED	The list of asynchronous task IDs should include those tasks which have completed recently.

Return Value

If the function is successful, the return value is the number of task identifiers copied into the provided array. If there are no tasks which match the requested criteria, the return value is zero. A return value of FTP_ERROR indicates an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpEnumTasks** function can be used to obtain a list of numeric identifiers that represent the asynchronous tasks that have been started or those that have completed. These task IDs are used by other functions to reference the background worker thread that has been created and obtain status information for the task. For example, the **FtpTaskDone** function can be used to determine if a

particular task has completed, and the **FtpTaskWait** function can be used to wait for a task to complete and return an error status code if the background operation failed.

There is an internal limit of 128 asynchronous tasks per process that may be active at any one time. When a task completes, the status information about that task is maintained for period of time after the task has completed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskDone, FtpTaskResume, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEventProc Function

```
VOID CALLBACK FtpEventProc(
   HCLIENT hClient,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **FtpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **FtpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **FtpRegisterEvent** function. This callback function is also used by asynchronous tasks to notify the application when the task has started and completed. The **FtpEventProc** function is a placeholder for the application-defined function name. Other functions, such as **FtpStartQueue** can also use event callbacks.

If the callback function is invoked by an asynchronous task, it will execute in the context of the worker thread that is managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself, and you should avoid calling any functions which could cause the thread to block. For example, you should not attempt to establish other network connections from within the event handler.

If your application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function for an asynchronous task. Controls should only be modified by the same UI thread that created their window. One common approach to resolve this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpDisableEvents, FtpEnableEvents, FtpFreezeEvents, FtpRegisterEvent, FtpStartQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpFindQueuedFile Function

```
DWORD WINAPI FtpFindQueuedFile(
    HQUEUE hQueue,
    LPCTSTR lpszFileName,
    DWORD dwQueueMode,
    DWORD dwExcludeFiles
);
```

The **FtpFindQueuedFile** function finds a file in the transfer queue which matches the search criteria.

Parameters

hQueue

A handle to a file transfer queue.

lpszFileName

A null-terminated string which specifies the name of the file to search for. The string may contain the wildcard character '?' to match against any single character, and '*' will match any number of characters. This parameter cannot be NULL or point to an empty string.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. This is a bitmask which may be one or more of the following values:

Constant	Description
FTP_QUEUE_ALL	Match the file name to all files in the queue.
FTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
FTP_QUEUE_UPLOAD	Match file names which are queued for upload.
FTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_QUEUE_FLAG_NONE	No files should be excluded from the search.
FTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
FTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
FTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **FtpCancelQueuedFile**

	function has been called and a queued file is in the process of being uploaded or downloaded.

Return Value

If the function succeeds, the return value is a unique identifier for the file. To obtain information about the file, call the **FtpGetQueuedFile** function. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function returns a unique file identifier for the first file which matches the specified file name in the queue. If the *lpszFileName* parameter includes wildcard characters, this function will return the first file in the queue which matches the name. File name matches are not case-sensitive, even when matching against remote file names on servers which use case-sensitive naming, such as UNIX based servers.

To perform more complex searches based on the file name, or to find multiple files, you can use the **FtpGetFirstQueuedFile** and **FtpGetNextQueuedFile** functions to iterate through all queued files.

Example

```
// Find a local file in the download queue
DWORD dwFileId = FtpFindQueuedFile(
                    hQueue,
                    lpszFileName,
                    FTP_QUEUE_DOWNLOAD | FTP_QUEUE_LOCAL,
                    FTP_QUEUE_FLAG_NONE);

// Display information about the queued file
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;

    if (FtpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpEnumQueuedFiles, FtpGetFirstQueuedFile, FtpGetNextQueuedFile, FtpGetQueuedFile, FtpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpFreezeEvents Function

```
INT WINAPI FtpFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **FtpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpDisableEvents, FtpEnableEvents, FtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetActivePorts Function

```
INT WINAPI FtpGetActivePorts(
   HCLIENT hClient,
   UINT *lpnLowPort,
   UINT *lpnHighPort
);
```

The **FtpGetActivePorts** function returns the local port numbers used for active mode file transfers.

Parameters

hClient

Handle to the client session.

lpnLowPort

Points to an unsigned integer that will contain the low port number when the function returns.

lpnHighPort

Points to an unsigned integer that will contain the high port number when the function returns.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to determine the default port numbers being used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **FtpSetActivePorts** function can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpSetActivePorts, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetAutoFileType Function

```
UINT WINAPI FtpGetAutoFileType(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   BOOL bScanFile
);
```

The **FtpGetAutoFileType** function returns the file transfer type based on the file extension or content.

Parameters

hClient

Handle to the client session. If this parameter is INVALID_CLIENT the function will only check against the global list of default text file extensions. If the handle is valid, it will also check for any application defined file extensions.

lpszFileName

A null-terminated string which specifies the local path to the file. This parameter cannot be NULL and cannot specify a local device name or directory.

bScanFile

Specifies if the contents of the file should be scanned. A value of zero indicates that only the file extension should be used to determine the file type, while a non-zero value specifies the contents of the file should be examined if the file type cannot be determined based on its extension.

Return Value

If the function succeeds, the return value is the file transfer type. If the function fails, the return value is INVALID_FILE_TYPE. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to determine the file transfer type to be used when uploading or downloading files. This function is called internally when FILE_TYPE_AUTO is specified as the default file type. The return value may be one of the following:

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-

	line character sequences will be preserved.

If the file extension or contents are not recognized, the default file transfer type for the client session will be returned. This will usually be FILE_TYPE_IMAGE, however this can be changed by calling the **FtpRegisterFileType** function. The file type for the current client session can be explicitly set using the **FtpSetFileType** function.

If the ***bScanFile*** parameter is non-zero, the local file will be opened in a shared reading mode and up to 4,096 bytes will be examined to determine if it contains binary data. If the file is currently locked or has been opened exclusively by another process, the file type associated with the file extension will be returned instead. Text files which contain UTF-16 text will always return a file type of FILE_TYPE_IMAGE because they can contain non-ASCII characters and/or embedded null characters.

If the ***bScanFile*** parameter is non-zero and the file type cannot be determined based on the file name extension, the file specified by ***lpszFileName*** must exist and be a regular file. If the file does not exist, an error will be returned and the last error code will be set to ST_ERROR_FILE_NOT_FOUND. If the ***bScanFile*** parameter is zero, no errors will be returned if the file does not exist, the function will only check the file name extension to determine the file type. When downloading a file, the ***bScanFile*** parameter should normally be zero because the local file may not exist yet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFileType, FtpRegisterFileType, FtpSetFileMode, FtpSetFileStructure, FtpSetFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetBufferSize Function

```
INT WINAPI FtpGetBufferSize(
    HCLIENT hClient
);
```

The **FtpGetBufferSize** function returns the size in bytes of an internal buffer that will be used during data transfers.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the size of the internal buffer that will be used in data transfers. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a value of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpSetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetChannelMode Function

```
INT WINAPI FtpGetChannelMode(
    HCLIENT hClient,
    INT nChannel
);
```

The **FtpGetChannelMode** function returns the mode of the specified communications channel.

Parameters

hClient

Handle to the client session.

nChannel

An integer value which specifies which channel to return information for. It may be one of the following values:

Value	Description
FTP_CHANNEL_COMMAND	Return information about the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Return information about the data channel. This is the communication channel used to send or receive data during a file transfer.

Return Value

If the function succeeds, the return value is the mode for the specified channel. If the function fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetChannelMode** function returns a integer which may be one of the following values:

Value	Description
FTP_CHANNEL_CLEAR	The channel is not encrypted. This is the default mode for both channels when a standard, non-secure connection is established with the server.
FTP_CHANNEL_SECURE	The channel is encrypted. This is the default mode for both channels when a secure connection is established with the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpSetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientQuota Function

```
INT WINAPI FtpGetClientQuota(
   HCLIENT hClient,
   LPFTPCLIENTQUOTA lpClientQuota
);
```

The **GetClientQuota** function returns information about file quotas for the current client session.

Parameters

hClient

Handle to the client session.

lpClientQuota

A pointer to an FTPCLIENTQUOTA structure which contains the quota information returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the XQUOTA command to obtain information for the current client session. If the server does not support this command, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetServerInformation, FtpGetServerType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetData Function

```
INT WINAPI FtpGetData(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved
);
```

The **FtpGetData** function transfers the contents of a file on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetData** function is used to download the contents of a remote file into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpdwLength*** parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when downloading the file, the variable

passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

The **FtpGetText** function can be used to download a text file and store the contents in a string.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = FtpGetData(hClient,
                     lpszRemoteFile,
                     &hgblBuffer,
                     &cbBuffer,
                     0);

if (nResult != FTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetFile, FtpGetText, FtpGetTransferStatus, FtpPutData, FtpPutFile, FtpRegisterEvent, FtpSetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetDefaultZoneId Function

```
BOOL WINAPI FtpGetDefaultZoneId(
    HCLIENT hClient,
    LPDWORD lpdwZoneId
);
```

The **FtpGetDefaultZoneId** function returns the default security zone for downloaded files.

Parameters

hClient

Handle to the client session.

lpdwZoneId

A pointer to an unsigned integer which is updated with the default zone ID for the client session. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the function succeeds, the return value is non-zero and *lpdwZoneId* will be updated with the default zone identifier. If no default zone has been explicitly set, the function will still return success, and the variable *lpdwZoneId* points to will be set to `SECURITY_ZONE_NONE`. If the function fails, it returns zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetDefaultZoneId** function returns the zone identifier which was previously set using the **FtpSetDefaultZoneId** function. If no default zone has been specified for the client session, the default behavior is to not set a security zone for the file. This maintains backwards compatibility with previous versions of the library which would not create the security zone metadata for local files.

This function is typically used to confirm the currently configured zone before downloading a file, or to verify whether the default behavior has been overridden. The default zone ID is specific to the client session and does not affect other sessions or system-wide settings.

To get the security zone identifier for a local file, use the **FtpGetFileZoneInfo** function. To change or remove the security zone information for a file, use the **FtpSetFileZoneInfo** function.

For more detailed information about the zone identifiers, refer to the **FtpSetDefaultZoneId** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetFileZoneInfo, FtpGetSecurityZoneId, FtpSetDefaultZoneId, FtpSetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetDirectory Function

```
INT WINAPI FtpGetDirectory(
   HCLIENT hClient,
   LPTSTR lpszDirectory,
   INT nMaxLength
);
```

The **FtpGetDirectory** function copies the current working directory on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a buffer that will contain the name of the current working directory on the server. The file pathing and name conventions must be that of the server.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null-character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the PWD command to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpCreateDirectory, FtpRemoveDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetDirectoryFormat Function

```
INT WINAPI FtpGetDirectoryFormat(
    HCLIENT hClient
);
```

The **FtpGetDirectoryFormat** function returns an identifier which specifies what format is being used by the server to list files. By default, the library will automatically determine the appropriate format, but this value may be overridden by the **FtpSetDirectoryFormat** function.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the currently selected directory format. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Refer to the **FtpSetDirectoryFormat** function for a list of directory format types that are supported by the library. This function can be used to determine which format was selected by the library after a file listing has been retrieved.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCloseDirectory, FtpGetFileStatus, FtpGetFirstFile, FtpGetNextFile, FtpOpenDirectory, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetErrorString Function

```
INT WINAPI FtpGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **FtpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **FtpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetLastError, FtpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFeatures Function

```
DWORD WINAPI FtpGetFeatures(
    HCLIENT hClient
);
```

The **FtpGetFeatures** function returns the server features available to the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is one or more bit flags which specify the features that are available to the client. If the function fails, it will return zero. Because it is possible that no features would be enabled, a return value of zero does not always indicate an error. An application should call **FtpGetLastError** to determine if an error code has been set.

Remarks

The **FtpGetFeatures** function returns a value which may be a combination of one or more of the following bit flags:

Value	Description
FTP_FEATURE_SIZE	The server supports the SIZE command to determine the size of a file. If this feature is not enabled, the library will attempt to use the MLST or STAT command to determine the file size.
FTP_FEATURE_STAT	The server supports using the STAT command to return information about a specific file. If this feature is not enabled, the client may not be able to obtain information about a specific file such as its size, permissions or modification time.
FTP_FEATURE_MDTM	The server supports the MDTM command to obtain information about the modification time for a specific file. This command may also be used to set the file time on the server.
FTP_FEATURE_REST	The server supports restarting file transfers using the REST command. If this feature is not enabled, the client will not be able to restart file transfers and must upload or download the complete file.
FTP_FEATURE_SITE	The server supports site specific commands using the SITE command. If this feature is not enabled, no site specific commands will be sent to the server.
FTP_FEATURE_IDLE	The server supports setting the idle timeout period using the SITE IDLE command to specify the number of seconds that the client may idle before the server terminates the connection.
FTP_FEATURE_CHMOD	The server supports modifying the permissions of a specific file using the SITE CHMOD command. If this feature is not enabled, the client will not be able to set the permissions for a file.
FTP_FEATURE_AUTH	The server supports explicit TLS sessions using the AUTH

	command. If this feature is not enabled, the client will only be able to connect to a secure server that uses implicit TLS connections. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PBSZ	The server supports the PBSZ command which specifies the buffer size used with secure data connections. If this feature is disabled, it may prevent the client from changing the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PROT	The server supports the PROT command which specifies the protection level for the data channel. If this feature is disabled, the client will be unable to change the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_CCC	The server supports the CCC command which returns the command channel to a non-secure mode. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_HOST	The server supports the HOST command which enables a client to specify the hostname after establishing a connection with a server that supports virtual hosting.
FTP_FEATURE_MLST	The server supports the MLST command which returns status information for files. If this feature is enabled, the MLST command will be used instead of the STAT command.
FTP_FEATURE_MFMT	The server supports the MFMT command which is used to change the last modification time for a file. If this command is supported, it is used instead of the MDTM command to change the modification time for a file.
FTP_FEATURE_XCRC	The server supports the XCRC command which returns the CRC-32 checksum for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_XMD5	The server supports the XMD5 command which returns an MD5 hash for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_LANG	The server supports the LANG command which sets the language used for the current client session. Command responses and file naming conventions will use the specified language.
FTP_FEATURE_UTF8	The server supports the OPTS UTF-8 command which specifies UTF-8 encoding when specifying filenames. This feature is typically used in conjunction with setting the default language for the client session.
FTP_FEATURE_XQUOTA	The server supports the XQUOTA command which returns quota information for the current client session.
FTP_FEATURE_UTIME	The server supports the UTIME command which is used to change the last modification time for a specified file.

When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client.

For example, the first time an application calls the **FtpGetFileSize** function to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpEnableFeature, FtpSetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFile Function

```
INT WINAPI FtpGetFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset
);
```

The **FtpGetFile** function transfers the specified file on the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be downloaded to the local system. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetMultipleFiles, FtpGetTransferStatus, FtpPutFile, FtpPutMultipleFiles, FtpRegisterEvent, FtpSetBufferSize, FtpVerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileEx Function

```
INT WINAPI FtpGetFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset
);
```

The **FtpGetFileEx** function transfers the specified file on the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be downloaded to the local system. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetMultipleFiles, FtpGetTransferStatusEx, FtpPutFile, FtpPutMultipleFiles, FtpRegisterEvent, FtpSetBufferSize, FtpVerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileList Function

```
INT WINAPI FtpGetFileList(
   HCLIENT hClient,
   LPCTSTR lpszDirectory,
   DWORD dwOptions,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **FtpGetFileList** function returns an unparsed list of files in the specified directory.

Parameters

hClient

Handle to the client session.

lpszDirectory

A null-terminated string that specifies the name of a directory and/or a wildcard file mask. The format of the directory name must match the file naming conventions of the server. If this parameter is NULL or points to an empty string, the current working directory will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_LIST_DEFAULT	This option specifies the server should return a complete listing of files in the specified directory with as much detail as possible. This typically means that the file size, date, ownership and access rights will be returned to the client. Information about the files are returned in lines of text, with each line terminated by carriage return and linefeed (CRLF) characters. The exact format of the data returned is specific to the server operating system.
FTP_LIST_NAMEONLY	This option specifies the server should only return a list of file names, with no additional information about the file. Each file name is terminated by carriage return and linefeed (CRLF) characters.

lpszBuffer

A null-terminated string buffer that will contain the list of files when the function returns. This buffer should be large enough to store the complete file listing and a terminating null character. If the buffer is smaller than the total amount of data returned by the server, the data will be truncated. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null character. If the function fails, the return value is FTP_ERROR. To get

extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetFileList** function returns a list of files in the specified directory, copying the data to a string buffer. Unlike the other functions like **FtpEnumFiles** that parse a directory listing and return information in an **FTPFILESTATUS** structure, this function returns the unparsed file list data. The actual format of the data that is returned depends on the operating system and how the server implements file listings. For example, UNIX servers typically return the output from the **/bin/ls** command.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **FtpChangeDirectory** function and then call this function again, passing NULL or an empty string as the *lpszDirectory* parameter.

This function can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the library. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the FTP_LIST_NAMEONLY option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This function is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
TCHAR szFileList[MAXFILELISTSIZE];

nResult = FtpGetFileList(hClient, NULL, FTP_LIST_NAMEONLY,
                         szFileList, MAXFILELISTSIZE);

if (nResult != FTP_ERROR)
    _tprintf(_T("%s\n"), szFileList);
else
{
    DWORD dwError = FtpGetLastError();
    _tprintf(_T("Unable to list files (error 0x%08lx)\n"), dwError);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpEnumFiles, FtpGetDirectoryFormat, FtpGetFirstFile, FtpOpenDirectory, FtpGetNextFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileNameEncoding Function

```
INT WINAPI FtpGetFileNameEncoding(
    HCLIENT hClient
);
```

The **FtpGetFileNameEncoding** function returns an identifier which specifies what type of encoding is being used when file names are sent to the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the type of encoding that is used. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Refer to the **FtpSetFileNameEncoding** function for a list of encoding types that are supported by the library.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCommand, FtpEnableFeature FtpGetFeatures, FtpSetFileNameEncoding, FtpSetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFilePathUrl Function

```
INT WINAPI FtpGetFilePathUrl(
   HCLIENT hClient,
   LPCTSTR lpszFilePath,
   LPTSTR lpszFileUrl,
   INT nMaxLength
);
```

The **FtpGetFilePathUrl** function returns a normalized URL for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFilePath

A null-terminated string which specifies the file path on the server. The path may be absolute or relative, and the path does not have to exist on the server. This parameter cannot be NULL.

lpszFileUrl

A null-terminated string buffer that will contain the URL for the specified file. If this parameter is NULL, nothing will be copied but the function will return the length of the URL, not including the terminating null character.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. This must include space for the terminating null character. If ***lpszFileUrl*** is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the ***lpszFileUrl*** buffer, not including the terminating null character. If a buffer is provided, it must be large enough to contain the entire URL or the function will fail. If the ***lpszFileUrl*** parameter is NULL, the function will return the length of the URL and nothing will be copied. If the function fails, it will return zero. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpGetFilePathUrl** function will return the URL associated with the specified file path. This can be useful if you wish to display or store URLs rather than relative paths to a file on the current server. For example, if you are connected to the server `example.com` on port 21 using a standard connection and you provide a relative file path such as `documents/invoice.pdf` this function would return a URL which looks like `ftp://example.com/documents/invoice.pdf`. For security reasons, the resulting URL will never contain user credentials.

The file path in the URL will be normalized to remove extraneous directory traversals and path separators. If the path is relative, it will be expanded to an absolute path based on the client's current working directory. If an absolute path is provided, it will be included in the URL as-is. It is not required for the file path to exist on the server at the time this function is called. For more information about path normalization, refer to the **FtpNormalizePath** function.

This function can also be used to obtain a URL which can be passed to the **FtpGetSercurityZoneId** function to determine which security zone should be assigned to the file when it is downloaded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

Example

The following example shows how **FtpGetFilePathUrl** could be used to obtain the origin URL for a file on the server and then use that in conjunction with the **FtpGetSecurityZoneId** function to update the security zone metadata for the file if the download is successful.

```
LPCTSTR lpszLocalFile = _T("\\Users\\Public\\Documents\\Document.pdf");
LPCTSTR lpszRemoteFile = _T("files/document.pdf");

TCHAR szFileUrl[FTP_MAXURLPATHLEN] = { 0 };
DWORD dwZoneId = SECURITY_ZONE_NONE;

// Return a complete URL for the relative file path specified for the
// remote file on the server
if (FtpGetFilePathUrl(hClient, lpszRemoteFile, szFileUrl, FTP_MAXURLPATHLEN) < 1)
{
    _tprintf(_T("Unable to get URL for \"%s\"\n"), lpszRemoteFile);
    return;
}

// Check the security zone for the remote file URL
if (!FtpGetSecurityZoneId(szFileUrl, &dwZoneId))
{
    _tprintf(_T("Unable to determine a security zone for \"%s\"\n"), szFileUrl);
}

// Download the file from the server
INT nResult = FtpGetFile(hClient, lpszLocalFile, lpszRemoteFile, 0, 0);

if (nResult != FTP_ERROR)
{
    // The download was successful; if a security zone was returned for
    // the file, then update it with the zone ID and origin URL
    if (dwZoneId != SECURITY_ZONE_NONE)
    {
        if (!FtpSetFileZoneInfo(lpszLocalFile, dwZoneId, szFileUrl))
        {
            _tprintf(_T("Unable to set the security zone for \"%s\"\n"),
lpszLocalFile);
            return;
        }
    }
}
else
{
    // The download failed
    _tprintf(_T("Unable to download \"%s\" to \"%s\"\n"), lpszRemoteFile,
lpszLocalFile);
}
```

See Also

FtpConnectUrl, FtpGetSecurityZoneId, FtpNormalizePath, FtpNormalizeUrl, FtpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFilePermissions Function

```
INT WINAPI FtpGetFilePermissions(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPDWORD lpdwPermissions
);
```

The **FtpGetFilePermissions** function returns information about the access permissions for a specific file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string which contains the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

lpdwPermissions

A pointer to an unsigned integer which will contain the access permissions for the file when the function returns. The file permissions are represented as bit flags, and may be one or more of the following values:

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.

FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the function succeeds, the return value is a result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the STAT command to retrieve information about the specified file. On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the function will fail and value pointed to by the ***lpdwPermissions*** parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFileStatus, FtpSetFilePermissions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileSize Function

```
INT WINAPI FtpGetFileSize(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPDWORD lpdwFileSize
);
```

The **FtpGetFileSize** function returns the size of the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the function call will fail, and the *lpdwFileSize* parameter will be set to zero.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Some FTP servers will refuse to return the size of a file if the current file type is set to FILE_TYPE_ASCII because the size of a text file on the server may not accurately reflect what the size of the file will be on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFileStatus, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileSizeEx Function

```
INT WINAPI FtpGetFileSizeEx(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   ULARGE_INTEGER *lpuiFileSize
);
```

The **FtpGetFileSizeEx** function returns the size of the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpuiFileSize

Points to a ULARGE_INTEGER structure that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the function call will fail, and the *lpdwFileSize* parameter will be set to zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFileStatusEx, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileStatus Function

```
BOOL WINAPI FtpGetFileStatus(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPFTPFILESTATUS lpFileStatus
);
```

The **FtpGetFileStatus** function returns information about a specific file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string which contains the name of the file that status information will be returned on. The file name cannot contain any wildcard characters.

lpFileStatus

A pointer to an FTPFILESTATUS structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the STAT command to retrieve information about the specified file. Unlike the **FtpGetFirstFile** and **FtpGetNextFile** functions, which read through a file list returned on the data channel, this function reads the result of a command string. For applications that need information about a specific file, using this function can be considerably faster than iterating through all of the files in a given directory. Note that not all servers support using the command in this way.

On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the FTPFILESTATUS structure members will be empty strings and zero values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFirstFile, FtpGetNextFile, FtpGetTransferStatus, FtpOpenDirectory, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileStatusEx Function

```
BOOL WINAPI FtpGetFileStatusEx(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPFTPFILESTATUSEX lpFileStatus
);
```

The **FtpGetFileStatusEx** function returns information about a specific file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string which contains the name of the file that status information will be returned on. The file name cannot contain any wildcard characters.

lpFileStatus

A pointer to an FTPFILESTATUSEX structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the STAT command to retrieve information about the specified file. Unlike the **FtpGetFirstFileEx** and **FtpGetNextFileEx** functions, which read through a file list returned on the data channel, this function reads the result of a command string. For applications that need information about a specific file, using this function can be considerably faster than iterating through all of the files in a given directory. Note that not all servers support using the command in this way.

On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the FTPFILESTATUSEX structure members will be empty strings and zero values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFirstFileEx, FtpGetNextFileEx, FtpGetTransferStatusEx, FtpOpenDirectory, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileTime Function

```
INT WINAPI FtpGetFileTime(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPSYSTEMTIME lpFileTime,
   BOOL bLocalize
);
```

The **FtpGetFileTime** function returns the modification time for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a SYSTEMTIME structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetFileTime** function can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time. If you plan on changing the values returned in the SYSTEMTIME structure and then calling **FtpSetFileTime** function to modify the file time on the server, you should do not localize the time.

This function uses the MDTM command to determine the modification time of the specified file. If the server does not support this command, the function will attempt to use the STAT command to determine the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFileStatus, FtpOpenDirectory, FtpSetFileTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileType Function

```
INT WINAPI FtpGetFileType(
   HCLIENT hClient,
   UINT *lpnFileType
);
```

The **FtpGetFileType** function returns the default file type for the current client session.

Parameters

hClient

Handle to the client session.

lpnFileType

A pointer to an unsigned integer that will identify the current file type. This parameter cannot be NULL. When the function returns it will contain one of the following values:

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **FtpRegisterFileType** function.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executables. If the type of file cannot be automatically determined, it will always be considered a binary file.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If this function is called when connected to an SFTP (SSH) server, the default file type will always be FILE_TYPE_IMAGE because SFTP does not differentiate between text files and binary files.

The **FtpSetFileType** function can be used to change the default file type. To determine the automatic file type for a specific file based on its extension or contents, use the **FtpGetAutoFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetAutoFileType, FtpOpenFile, FtpRegisterFileType, FtpSetFileMode, FtpSetFileStructure, FtpSetFileType, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFileZoneInfo Function

```
BOOL WINAPI FtpGetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   LPDWORD lpdwZoneId,
   LPTSTR lpszHostUrl,
   INT nMaxLength
);
```

The **FtpGetFileZoneInfo** function returns the security zone information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and read from the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

lpdwZoneId

A pointer to an unsigned integer which will contain the zone ID assigned to the file when the function returns. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string buffer which will contain the URL used to originally download the file. This information is not always available, in which case an empty string will be returned. If this information is not required, this parameter may be NULL and the ***nMaxLength*** parameter should be specified with a value of zero.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the ***lpszHostUrl*** string including the terminating null character. If the ***lpszHostUrl*** parameter is NULL, this value should be zero. If the buffer is not large enough to contain the entire URL, the string will be truncated. The maximum allowed URL length is 2,048 characters.

Return Value

If the function succeeds, the return value is non-zero. If the function fails or zone information is not available, the function returns zero. To get extended error information, call **FtpGetLastError**.

Remarks

When a file is downloaded from the Internet or other external sources, Windows may assign a zone identifier to the file using an alternate data stream named `Zone.Identifier`. This metadata helps determine how the system treats the file with respect to security, such as whether to prompt the user with a warning before opening it.

The **FtpGetFileZoneInfo** function retrieves the zone ID associated with the specified local file, allowing an application to determine the origin and trust level of that file. This can be useful when implementing additional security measures, auditing downloaded content, or providing user warnings about potentially untrusted files.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be read from files on network volumes because the SMB protocol does not support alternate data streams.

In addition to the zone ID, this function can optionally return the source URL used to download the file. This information, if present, is extracted from the same alternate data stream. Not all files will include this URL, as it depends on the program or browser that downloaded the file and whether it preserved this metadata. This function will not return an error if the URL stored with the security zone information is not a valid FTP URL. For example, it's possible for this function to return zone information with an HTTP URL. You can use the **FtpValidateUrl** function to determine if the host URL is a valid FTP URL.

Note that not all file systems support alternate data streams. For example, FAT32 volumes do not support them, and in such cases, zone information will not be available. Additionally, some applications may remove this metadata after download or strip it during file copy operations. The absence of zone information does not guarantee that a file is safe or originated locally.

If you wish to distinguish between a file which exists without any zone information, and an error condition where the file does not exist or could not be accessed, check the last error code. If this function returns zero and the **FtpGetLastError** function returns NO_ERROR, this indicates the file name is valid and it exists, but the `Zone.Identifier` alternate data stream does not exist.

For more detailed information about the security zone identifiers, refer to the **FtpSetDefaultZoneId** function. To change or remove the security zone metadata from a local file, use the **FtpSetFileZoneInfo** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpDownloadFile, FtpGetDefaultZoneId, FtpSetDefaultZoneId, FtpSetFileZoneInfo, FtpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFirstFile Function

```
BOOL WINAPI FtpGetFirstFile(
   HCLIENT hClient,
   LPFTPFILESTATUS lpFileStatus
);
```

The **FtpGetFirstFile** function returns the first file in the directory listing returned by the server after a call to the **FtpOpenDirectory** function.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an FTPFILESTATUS structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This file list information returned by the server is cached by the library, allowing you to use this function to reposition back to the beginning of the file list.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)
{
    FTPFILESTATUS ftpFile;
    BOOL bResult;

    bResult = FtpGetFirstFile(hClient, &ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = FtpGetNextFile(hClient, &ftpFile);
    }

    FtpCloseDirectory(hClient);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFileStatus, FtpGetNextFile, FtpOpenDirectory, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFirstFileEx Function

```
BOOL WINAPI FtpGetFirstFileEx(
   HCLIENT hClient,
   LPFTPFILESTATUSEX lpFileStatus
);
```

The **FtpGetFirstFileEx** function returns the first file in the directory listing returned by the server after a call to the **FtpOpenDirectory** function. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an FTPFILESTATUSEX structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This file list information returned by the server is cached by the library, allowing you to use this function to reposition back to the beginning of the file list.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)
{
    FTPFILESTATUSEX ftpFile;
    BOOL bResult;

    bResult = FtpGetFirstFileEx(hClient, &ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = FtpGetNextFileEx(hClient, &ftpFile);
    }

    FtpCloseDirectory(hClient);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFileStatusEx, FtpGetNextFileEx, FtpOpenDirectory,

FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetFirstQueuedFile Function

```
DWORD WINAPI FtpGetFirstQueuedFile(
   HQUEUE hQueue,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPFTPQUEUEDFILE lpFileInfo
);
```

The **FtpGetFirstQueuedFile** function returns information about the first file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. It may be one of the following values:

Constant	Description
FTP_QUEUE_ALL	Return information about all files in the queue.
FTP_QUEUE_DOWNLOAD	Return information for files which are queued for download.
FTP_QUEUE_UPLOAD	Return information for files which are queued for upload.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_QUEUE_FLAG_NONE	No files should be excluded.
FTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
FTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
FTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **FtpCancelQueuedFile** function has been called and a queued file is in the process of being uploaded or downloaded.

lpFileInfo

A pointer to a FTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no queued files which match the specified parameters, or the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function returns information about the first file in the current transfer queue. It is used in conjunction with the **FtpGetNextQueuedFile** function to obtain information about all queued files.

Example

```
// List all files in a file transfer queue
FTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = FtpGetFirstQueuedFile(hQueue, FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE,
&queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case FTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case FTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
    }

    dwFileId = FtpGetNextQueuedFile(hQueue, &queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpEnumQueuedFiles, FtpFindQueuedFile, FtpGetNextQueuedFile, FtpGetQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetIdleTimeout Function

```
INT WINAPI FtpGetIdleTimeout(
    HCLIENT hClient
);
```

The **FtpGetIdleTimeout** function returns the idle timeout period for the control channel.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetIdleTimeout** function returns the server-side idle timeout value that was most recently set for the control connection using the **FtpSetIdleTimeout** function. This value represents how long the server will wait during periods of inactivity before automatically closing the connection. If the idle timeout has not been explicitly set during the session, the function will return the default timeout value of 300 seconds (5 minutes).

The timeout value is only available if the server supports the non-standard `SITE IDLE` command. If the server does not support this command, this function will fail. In that case, the default timeout enforced by the server will apply, but the timeout period cannot be accessed or changed by the client. This function will also return an error if the session is using SFTP, as that protocol does not separate control and data channels and does not support idle timeout settings in the same way.

This function does not query the server for the current idle timeout value for the control channel. It simply returns the last known value that was successfully applied during the current session, if available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCommand, FtpGetTimeout, FtpSetIdleTimeout, FtpSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetLastError Function

```
DWORD WINAPI FtpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **FtpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **FtpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **FtpSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those functions which call **FtpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetErrorString, FtpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetMultipleFiles Function

```
INT WINAPI FtpGetMultipleFiles(
   HCLIENT hClient,
   LPCTSTR lpszLocalDirectory,
   LPCTSTR lpszRemoteDirectory,
   LPCTSTR lpszFileMask,
   DWORD dwReserved
);
```

The **FtpGetMultipleFiles** function copies one or more files from the server to the local host, using the specified wildcard.

Parameters

hClient

Handle to the client session.

lpszLocalDirectory

Pointer to a null-terminated string which specifies the local directory where the files will be downloaded to. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs. A NULL pointer or empty string specifies that files should be copied to the current working directory.

lpszRemoteDirectory

Pointer to a null-terminated string which specifies the remote directory where the files will be downloaded from. A NULL pointer or empty string specifies that the files should be copied from the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the server to the local system. The file mask should follow the native conventions used for wildcard file matches on the server.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetMultipleFiles** function is used to transfer files from the server to the local host which match a specified wildcard file mask. This function requires that the client be able to automatically list and parse directory listings from the server, otherwise an error will be returned. All files will be transferred using the current file type as specified by the **FtpSetFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpGetFile, FtpPutFile, FtpPutMultipleFiles, FtpSetFileType

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetNextFile Function

```
BOOL WINAPI FtpGetNextFile(
   HCLIENT hClient,
   LPFTPFILESTATUS lpFileStatus
);
```

The **FtpGetNextFile** function returns the next file in the directory listing returned by the server.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an FTPFILESTATUS structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetNextFile** function returns the next file in the directory listing. If the last file has been returned, the function will return zero and the client should call the **FtpCloseDirectory** function to close the directory.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)
{
    FTPFILESTATUS ftpFile;
    BOOL bResult;

    bResult = FtpGetFirstFile(hClient, &ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = FtpGetNextFile(hClient, &ftpFile);
    }

    FtpCloseDirectory(hClient);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFileStatus, FtpGetFirstFile, FtpOpenDirectory, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetNextFileEx Function

```
BOOL WINAPI FtpGetNextFileEx(
   HCLIENT hClient,
   LPFTPFILESTATUSEX lpFileStatus
);
```

The **FtpGetNextFileEx** function returns the next file in the directory listing returned by the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an FTPFILESTATUSEX structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetNextFileEx** function returns the next file in the directory listing. If the last file has been returned, the function will return zero and the client should call the **FtpCloseDirectory** function to close the directory.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)
{
    FTPFILESTATUSEX ftpFile;
    BOOL bResult;

    bResult = FtpGetFirstFileEx(hClient, &ftpFile);
    while (bResult)
    {
        // The ftpFile structure contains information about the file
        bResult = FtpGetNextFileEx(hClient, &ftpFile);
    }

    FtpCloseDirectory(hClient);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFileStatusEx, FtpGetFirstFileEx, FtpOpenDirectory,

FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetNextQueuedFile Function

```
DWORD WINAPI FtpGetNextQueuedFile(
   HQUEUE hQueue,
   LPFTPQUEUEDFILE lpFileInfo
);
```

The **FtpGetNextQueuedFile** function returns information about the next file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

lpFileInfo

A pointer to a FTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no additional queued files which match the exclusion criteria specified with the call to **FtpGetFirstQueuedFile**, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function returns information about the next file in the current transfer queue after an initial call to **FtpGetFirstQueuedFile**. To obtain a list of all matching files in the queue, call this function repeatedly until it returns a value of zero. When information about the last file in the queue has been returned, the last error code will be set to `ST_ERROR_END_OF_QUEUE`.

Example

```
// List all files in a file transfer queue
FTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = FtpGetFirstQueuedFile(hQueue, FTP_QUEUE_ALL, FTP_QUEUE_FLAG_NONE,
&queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case FTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case FTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
```

```
    }

    dwFileId = FtpGetNextQueuedFile(hQueue, &queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpEnumQueuedFiles, FtpFindQueuedFile, FtpGetFirstQueuedFile, FtpGetQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetPriority Function

```
INT WINAPI FtpGetPriority(
   HCLIENT hClient
);
```

The **FtpGetPriority** function returns a value which specifies the priority of file transfers.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current file transfer priority. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetPriority** function can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Value	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpSetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetProxyType Function

```
INT WINAPI FtpGetProxyType(
   HCLIENT hClient
);
```

The **FtpGetProxyType** function returns the type of proxy that the client is connected to. By default, no proxy server is specified and this function returns a value of FTP_PROXY_NONE. For a list of possible proxy server types, refer to the **FtpProxyConnect** function.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value identifies the type of proxy that the client is connected to. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetQueueClient Function

```
HCLIENT WINAPI FtpGetQueueClient(
    HQUEUE hQueue
);
```

The **FtpGetQueueClient** function returns a handle to the current client session.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value a handle to a client session. If the function fails, it will return INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetQueueClient** function returns the handle to a client session which may be used with other functions, such as **FtpGetTransferStatus**. A valid handle will only be returned if the queue manager is currently uploading or downloading a file. If the queue is active but there are no file transfers in progress at the time this function is called, it will fail and set the last error code to `ST_ERROR_NO_QUEUED_TRANSFER`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetTransferStatus, FtpGetTransferStatusEx, FtpGetQueueStatus, FtpGetThreadQueue, FtpStartQueue, FtpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetQueuedFile Function

```
BOOL WINAPI FtpGetQueuedFile(
   HQUEUE hQueue,
   DWORD dwFileId,
   LPFTPQUEUEDFILE lpFileInfo
);
```

The **FtpGetQueuedFile** function returns information about the specified file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

lpFileInfo

A pointer to a FTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the file identifier is not valid or the file has been removed from the queue, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The file identifier can be obtained by several different functions, including **FtpEnumQueuedFiles** and **FtpGetQueueStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

Example

```
// Get the number of files in the queue
LONG nFiles = FtpEnumQueuedFiles(hQueue, NULL, 0)

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];

    nFiles = FtpEnumQueuedFiles(hQueue, pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
        FTPQUEUEDFILE queuedFile;

        if (FtpGetQueuedFile(hQueue, pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
```

```
                _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                 break;

        case FTP_QUEUE_UPLOAD:
                _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                 break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpEnumQueuedFiles, FtpFindQueuedFile, FtpGetFirstQueuedFile, FtpGetNextQueuedFile, FtpGetQueueStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetQueueStatus Function

```
INT WINAPI FtpGetQueueStatus(
  HQUEUE hQueue,
  LPFTPQUEUESTATUS lpQueueStatus
);
```

The **FtpGetQueuedFile** function returns information about the specified file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

lpQueueStatus

A pointer to a FTPQUEUESTATUS structure which will contain information about the current state of the file transfer queue. If this parameter is NULL, the function will ignore the parameter and only return the current status of the queue.

Return Value

If the function succeeds, the return value is the current queue status. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**. The following status values may be returned:

Value	Description
FTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before **FtpStartQueue** has been called or after **FtpStopQueue** has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
FTP_QUEUE_STATUS_ACTIVE	The queue manager is active and files are currently being uploaded or downloaded.
FTP_QUEUE_STATUS_PAUSED	The queue manager is active although file transfers are currently paused. The queue enters this state after the **FtpSuspendQueue** function is called and resumes file transfers after the **FtpResumeQueue** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancelQueuedFile, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpSuspendQueue, FtpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetQueueThread Function

```
DWORD WINAPI FtpGetQueueThread(
    HQUEUE hQueue
);
```

The **FtpGetQueueThread** function returns a value which uniquely identifies the queue manager thread.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero and specifies a unique identifier for the queue manager thread. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetQueueThread** function can be used to obtain a unique identifier for the queue manager thread which is responsible for performing the file transfers. If the function returns zero, this indicates that either the queue handle is no longer valid, or the queue is idle and no files are currently being processed. To obtain the handle to a queue associated with a particular thread, call the **FtpGetThreadQueue** function.

You should not store the thread ID or consider it a persistent value, even during the lifetime of the process. It is only valid while the queue manager is actively processing files in the queue. The thread ID value will change every time the **FtpStartQueue** function is called.

Never use the thread ID to obtain a handle to the thread and call Windows API functions such as **SuspendThread** or **TerminateThread**. This can result inconsistency within the internal state of the queue manager and may result in incomplete or corrupted file transfers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetQueueStatus, FtpGetThreadQueue, FtpStartQueue, FtpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetResultCode Function

```
INT WINAPI FtpGetResultCode(
    HCLIENT hClient
);
```

The **FtpGetResultCode** function reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCommand, FtpGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetResultString Function

```
INT WINAPI FtpGetResultString(
   HCLIENT hClient,
   LPTSTR lpszResult,
   INT cbResult
);
```

The **FtpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

The buffer that will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server.

If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **FtpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCommand, FtpGetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetSecurityInformation Function

```
BOOL WINAPI FtpGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **FtpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (FtpGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
```

```
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpCreateSecurityCredentials, FtpDeleteSecurityCredentials, FtpDisconnect, FtpProxyConnect, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetSecurityZoneId Function

```
BOOL WINAPI FtpGetSecurityZoneId(
   LPCTSTR lpszResource,
   LPDWORD lpdwZoneId
);
```

The **FtpGetSecurityZoneId** function returns the security zone identifier for the specified resource.

Parameters

lpszResource

A null-terminated string which specifies the resource URL to check. This must be either a valid FTP URL or a local file URL.

lpdwZoneId

A pointer to an unsigned integer which is updated with the zone ID for the specified resource. It may be one of the following values when the function returns:

Value	Description
SECURITY_ZONE_NONE	The security zone for this resource cannot be determined.
SECURITY_ZONE_LOCAL	The resource is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The resource originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The resource originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The resource originates from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The resource originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the function succeeds, the return value is non-zero and ***lpdwZoneId*** will be updated with the zone identifier. If the security zone could not be determined, ***lpdwZoneId*** will be set to `SECURITY_ZONE_NONE`. If the function fails, it returns zero and the error code can be retrieved by calling **FtpGetLastError**.

Remarks

This **FtpGetSecurityZoneId** function is used to determine how a resource should be handled in terms of local security policy before performing a file transfer or download. For example, you may want to determine whether a resource originates from the Internet or local intranet before deciding whether to attach zone metadata, prompt the user, or apply additional security restrictions. It can also be used to implement policy-based rules for accepting or rejecting downloads based on their origin.

The ***lpszResource*** parameter must use the `ftp://`, `ftps://`, `ftpes://`, `sftp://` or `file://` schemes. This function will not accept URLs for other schemes such as `http://` or `https://`. While this function does not verify that the resource exists, it does require the URL to be syntatically valid. If the host name cannot be resolved to an IPv4 or IPv6 address, the function will fail. If the resource is a

local file, it must be a regular file and cannot be the name of a folder or device.

If the resource is specified using a `file://` URL that refers to a file on a network share (e.g.: `file://server/share/file.txt`), the function will attempt to access that file across the network. This may cause the application to block while Windows resolves the hostname and establishes a connection to the shared volume. If the network share is unavailable or slow to respond, the function may block until the connection succeeds or the attempt times out.

If this function returns FALSE, this indicates either the resource type is not supported or there was an internal failure when the Internet Security Manager attempted to map the resource URL to a security zone. In most default configurations, this function will return the zone `SECURITY_ZONE_INTERNET` for a valid URL unless the host has been explicitly added to the **Intranet Zone** or **Trusted Sites** list on the local system.

Security zones are used with Mark of the Web (MOTW) metadata which indicates if a file originated from an external source, such as the Internet or an untrusted network. When a file is downloaded using a web browser or other applications, it may add a `Zone.Identifier` alternate data stream to the file. This stream contains information such as `ZoneId=3`, which corresponds to the Internet Zone. Your application can use the **FtpGetFileZoneInfo** function to retrieve this information, which allows you to determine whether a file was downloaded from an external source and what zone Windows has associated with it. This metadata affects how files are handled by Windows security components such as Microsoft Defender, SmartScreen, Office Protected View, and script execution policies in Windows Explorer. If the file has no metadata, it is generally treated as coming from the Local Machine Zone (zone 0). If your application sets or clears this metadata using the **FtpSetFileZoneInfo** function, it can influence how other applications interpret the file's trust level.

It is important to note that this function is used to determine what security zone would be appropriate for a given resource, and it only performs syntactic checks and classification of the URL using the **IInternetSecurityManager** COM interface. The security zone this function returns is based on the current system and user policy, using zone assignments stored in the registry. Typically this information is managed using Windows settings or Internet options in the control panel. This function does not read the zone identifier metadata which has been assigned to a file.

For more detailed information about the zone identifiers, refer to the **FtpSetDefaultZoneId** function.

Example

```
LPCTSTR pszDownloadUrl = _T("ftp://example.com/pub/download/file.txt");
DWORD dwZoneId = SECURITY_ZONE_NONE;

if (FtpGetSecurityZoneId(pszDownloadUrl, &dwZoneId))
{
    // dwZoneId now contains SECURITY_ZONE_INTERNET
    _tprintf(_T("The security zone is: %u\n"), dwZoneId);
}
else
{
    DWORD dwError = FtpGetLastError();
    // Handle the error
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetDefaultZoneId, FtpGetFileZoneInfo, FtpSetDefaultZoneId, FtpSetFileZoneInfo, FtpValidateUrl

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerInformation Function

```
INT WINAPI FtpGetServerInformation(
   HCLIENT hClient,
   LPTSTR lpszSystemInfo,
   INT nMaxLength
);
```

The **FtpGetServerInformation** function returns information about the server, typically including the operating system type, version and platform.

Parameters

hClient

Handle to the client session.

lpszSystemInfo

The buffer that will contain the system information returned by the server.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SYST command to the server. The first word will identify the type of operating system. The format for the remaining information depends on the server type. Typically it is a description of the operating system version and hardware platform. For example, here are some strings commonly returned by various FTP servers:

Example	Description
UNIX Type: L8	A standard UNIX based server. This is the most common value returned by servers, and this indicates that the server supports UNIX file naming and directory listing conventions. This string may also include additional information such as the specific variant of UNIX and its version. The L8 portion of the string is a convention that lets the client know that a byte consists of 8 bits.
Windows_NT Version 5.0	A standard Windows based server, typically part of Internet Information Services (ISS). The server will use Windows file naming and directory listing conventions. The version identifies the specific release of Windows. For example, version 4.0 specifies Windows NT 4.0 and 5.0 specifies Windows 2000.
VMS V7.1 AlphaServer	A server running the VMS operating system. The server will use the standard file naming and directory listing conventions for that platform. Note that it is possible that a VMS system may also be configured to operate in a UNIX emulation mode, in which case it will return UNIX instead of VMS.
Netware	A server running the Netware operating system. The server will use the

system type	standard file naming and directory listing conventions for that platform. Note that it is possible that a Netware system may be configured to operate in a UNIX emulation mode, in which case it return UNIX instead of Netware.
WORLDGROUP Type: L8	A server running the WorldGroup software on the Windows platform. This server supports UNIX file naming and directory listing conventions. WorldGroup is a collaborative workgroup, email and file sharing service which includes an FTP server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpGetDirectoryFormat, FtpGetClientQuota, FtpGetServerStatus, FtpGetServerTimeZone, FtpGetServerType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerStatus Function

```
INT WINAPI FtpGetServerStatus(
   HCLIENT hClient,
   LPTSTR lpszStatus,
   INT nMaxLength
);
```

The **FtpGetServerStatus** function requests that the server return status information about itself.

Parameters

hClient

Handle to the client session.

lpszStatus

The buffer that will contain the system status returned by the server.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the STAT command to the server. The format for the information returned depends on the server type. Typically it is a description of the server platform, version, current user and file transfer options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerInformation, FtpGetServerType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerTimeZone Function

```
INT WINAPI FtpGetServerTimeZone(
   HCLIENT hClient,
   LPLONG lpnTimeZone
);
```

The **FtpGetServerTimeZone** function returns the timezone for the current server.

Parameters

hClient

Handle to the client session.

lpnTimeZone

A pointer to a signed long integer which will contain the timezone offset in seconds.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SITE ZONE command to the server to determine its timezone. The value returned is expressed as the number of seconds offset from Coordinated Universal Time (UTC). A positive value specifies a time west of UTC, while a negative value specifies a time east of UTC. For example, a value of 28800 would specify an offset of 8 hours west of UTC, which is the Pacific timezone.

The SIZE ZONE command is an extension that is not supported by all servers. If the server timezone cannot be determined, the function will fail and the value pointed to by the ***lpnTimeZone*** parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetClientQuota, FtpGetServerInformation, FtpGetServerStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerType Function

```
INT WINAPI FtpGetServerType(
    HCLIENT hClient
);
```

The **FtpGetServerType** function returns the type of server the client has connected to.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is a numeric value which indicates the server type. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SYST command to the server to determine the server type. The following server types are recognized by the library:

Value	Description
FTP_SERVER_UNKNOWN	The server type could not be determined by issuing the SYST command. The server may not support the command, or the command may only be allowed when issued by an authenticated user.
FTP_SERVER_UNIX	The server is running on a UNIX based operating system. This can include Linux and other variants, as well as operating systems which emulate UNIX style file pathing and directory listings.
FTP_SERVER_MSDOS	The server is running on an MS-DOS based operating system. The server expects file pathing and naming conventions according to the standard MS-DOS format and returns directory listings similar to the output of the DIR command.
FTP_SERVER_WINDOWS	The server is running on a Windows based operating system. The server expects file pathing and naming conventions according to the standard Windows long filename format, and returns directory listings similar to the output of the DIR command. Note that Windows servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a Windows platform.
FTP_SERVER_VMS	The server is running on a DEC VMS based operating system. The server expects file pathing and naming conventions specific to that operating system. Note that VMS servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a VMS platform.

FTP_SERVER_NETWARE	The server is running on a Netware based operating system. The server expects file pathing and naming conventions similar to the standard Windows long filename format, and returns directory listings that are similar to UNIX systems with the exception of the access and permissions flags for the file. Note that a Netware system may return listings in different formats based on the filesystem and site specific options specified.
FTP_SERVER_OTHER	The server type was not recognized. An attempt will be made to automatically determine the correct file pathing and naming conventions used by the server. To obtain a list of files on the server, it may be necessary to use the **FtpSetDirectoryFormat** function to specify the directory listing format.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetClientQuota, FtpGetServerInformation, FtpGetServerStatus, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetStatus Function

```
INT WINAPI FtpGetStatus(
   HCLIENT hClient
);
```

The **FtpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
FTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
FTP_STATUS_CONNECT	The client is establishing a connection with the server.
FTP_STATUS_READ	The client is reading data from the server.
FTP_STATUS_WRITE	The client is writing data to the server.
FTP_STATUS_DISCONNECT	The client is disconnecting from the server.
FTP_STATUS_OPENFILE	The client is opening a data connection to the server.
FTP_STATUS_CLOSEFILE	The client is closing the data connection to the server.
FTP_STATUS_GETFILE	The client is downloading a file from the server.
FTP_STATUS_PUTFILE	The client is uploading a file to the server.
FTP_STATUS_FILELIST	The client is retrieving a file listing from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session. To obtain status information about a file transfer, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpIsBlocking, FtpIsReadable, FtpIsWritable, FtpGetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetTaskError Function

```
DWORD WINAPI FtpGetTaskError(
    UINT nTaskId
);
```

Return the last error code for the specified asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed successfully, this function returns a value of zero. A non-zero return value indicates an error has occurred.

Remarks

The **FtpGetTaskError** function returns the last error code associated with the specified asynchronous task. If the task completed successfully, the return value will be zero. If the task is still active, the function will return the error ST_ERROR_TASK_ACTIVE. If the task has been suspended, the function will return ST_ERROR_TASK_SUSPENDED. Any other value indicates that the task completed, but the operation has failed and the error code will specify the cause of the failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskAbort, FtpTaskResume, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetTaskId Function

```
UINT WINAPI FtpGetTaskId(
   HCLIENT hClient
);
```

Return the asynchronous task identifier associated with the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the client handle is not associated with an asynchronous task, the function will return a value of zero.

Remarks

The **FtpGetTaskId** function will return the task ID that is associated with a client session. This is a unique unsigned integer value that references the worker thread that was created to manage the asynchronous client session. This function should only be called within an event handler that is invoked by a background task that has been started using a function such as **FtpAsyncGetFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskAbort, FtpTaskDone, FtpTaskResume, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetText Function

```
INT WINAPI FtpGetText(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **FtpGetText** function copies the contents of a text file on the server to the specified string buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A null-terminated string that specifies a text file on the server. The file pathing and naming conventions must be that of the host operating system.

lpszBuffer

A null-terminated string buffer which will contain the contents of the text file when the function returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetText** function is used to download a text file and store the contents in a string buffer. This function will always set the file type to FILE_TYPE_ASCII before downloading the file, and will restore the default file type before the method returns. Because binary files can include embedded null characters, this function should only be used with known text files.

This function has been included as a convenience for applications that need to retrieve relatively small text files and manipulate the contents as a string. If the Unicode version of this function is called, the contents of the text file is automatically converted to a Unicode string. If the size of the file is unknown or the text file is very large, it is recommended that you use the **FtpGetData** or **FtpGetFile** functions.

If you use the **FtpGetFileSize** function to determine how large the string buffer should be prior to calling this function, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **FtpGetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line characters.

FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = FtpGetText(hClient, lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetData, FtpGetFile, FtpGetTransferStatus, FtpPutData, FtpPutFile, FtpRegisterEvent, FtpSetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetThreadQueue Function

```
HQUEUE WINAPI FtpGetThreadQueue(
    DWORD dwThreadId
);
```

The **FtpGetThreadQueue** function returns a handle to the queue associated with the specified thread.

Parameters

dwThreadId

An unsigned integer value which specifies a unique identifier for the queue manager thread. If this value is zero, it will return the handle associated with the current thread.

Return Value

If the function succeeds, the return value is the handle to the queue. If the function fails, it will return INVAID_QUEUE. To get extended error information, call **FtpGetLastError**.

Remarks

All queued file transfers are handled by a queue manager which executes asynchronously in a background worker thread. This worker thread is created when the **FtpStartQueue** function is called and terminates when all files in the queue have been processed, or when the **FtpStopQueue** function is called. This function will return the queue handle associated with that worker thread.

The most common use of this function would be to obtain the handle for the queue from within a queue event handler. The application would pass in a value of zero as the thread ID and the function will return the handle to the queue which invoked the event notification.

If you call this function outside of an event handler and the ***dwThreadId*** parameter is zero, or if the queue is idle and not transferring files, this function will return INVALID_QUEUE. The thread ID is only valid while the queue manager is actively processing files in the queue.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetQueueStatus, FtpGetQueueThread, FtpStartQueue, FtpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetTimeout Function

```
INT WINAPI FtpGetTimeout(
   HCLIENT hClient
);
```

The **FtpGetTimeout** function returns the number of seconds the client will wait for a response from the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both commands issued on the control channel and file transfers using the data channel. The initial timeout value is set when establishing a connection and the default timeout period is 20 seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpIsReadable, FtpIsWritable, FtpProxyConnect, FtpRead, FtpSetTimeout, FtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetTransferStatus Function

```
INT WINAPI FtpGetTransferStatus(
   HCLIENT hClient,
   LPFTPTRANSFERSTATUS lpStatus
);
```

The **FtpGetTransferStatus** function returns information about the current file transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an **FTPTRANSFERSTATUS** structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetTransferStatus** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **FTPTRANSFERSTATUS** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FtpGetTransferStatusEx** function which returns the size as a 64-bit value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEnableEvents, FtpGetFileStatus, FtpGetTransferStatusEx, FtpRegisterEvent, FTPTRANSFERSTATUS, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetTransferStatusEx Function

```
INT WINAPI FtpGetTransferStatusEx(
   HCLIENT hClient,
   LPFTPTRANSFERSTATUS lpStatus
);
```

The **FtpGetTransferStatusEx** function returns information about the current file transfer in progress. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetTransferStatusEx** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **FTPTRANSFERSTATUSEX** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEnableEvents, FtpGetFileStatus, FtpGetTransferStatus, FtpRegisterEvent, FTPTRANSFERSTATUS, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpInitialize Function

```
BOOL WINAPI FtpInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **FtpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **FtpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **FtpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **FtpInitialize** or **FtpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpDisconnect, FtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsBlocking Function

```
BOOL WINAPI FtpIsBlocking(
   HCLIENT hClient
);
```

The **FtpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **FtpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancel, FtpGetStatus, FtpIsConnected, FtpIsReadable, FtpIsWritable, FtpRead, FtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsConnected Function

```
BOOL WINAPI FtpIsConnected(
   HCLIENT hClient
);
```

The **FtpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetStatus, FtpIsBlocking, FtpIsReadable, FtpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsReadable Function

```
BOOL WINAPI FtpIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **FtpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetStatus, FtpIsBlocking, FtpIsConnected, FtpIsWritable, FtpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsWritable Function

```
BOOL WINAPI FtpIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **FtpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetStatus, FtpIsBlocking, FtpIsConnected, FtpIsReadable, FtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpLogin Function

```
INT WINAPI FtpLogin(
   HCLIENT hClient,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszAccount
);
```

The **FtpLogin** function authenticates the specified user in on the server. This function must be called after the connection has been established, and before attempting to transfer files or perform any other function on the server.

Parameters

hClient

Handle to the client session.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

lpszAccount

Points to a string that specifies the account name to be used to authenticate the current client session. This parameter may be NULL or an empty string if no account name is required for the specified user.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Some public FTP servers support anonymous logins, where a username and password are not required to access the server. In this case, both the *lpszUserName* and *lpszPassword* parameters can be NULL or specify empty strings. In most cases, access to the server using an anonymous login is restricted, with clients only having permission to download files. Servers may also restrict the maximum number of anonymous sessions that may be logged in at one time.

This function should only be used after calling the **FtpLogout** function, enabling you to log in as another user during the same session. Not all servers will permit a client to change user credentials during the same session. In most cases, it is preferable to disconnect from the server and re-connect using the new credentials rather than using this function.

This function is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpLogout, FtpProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpLogout Function

```
INT WINAPI FtpLogout(
   HCLIENT hClient
);
```

The **FtpLogout** function logs out the user associated with the current client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpLogout** function is used when the client wants to re-authenticate using a new username and password. Before any further action may be taken, other than disconnecting from the server, the **FtpLogin** function must be called to re-authenticate the client.

It is not necessary to call this function prior to disconnecting from the server because the user current user is automatically logged out when the **FtpDisconnect** function is called.

This function is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpDisconnect, FtpLogin

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpMountStructure Function

```
INT WINAPI FtpMountStructure(
   HCLIENT hClient,
   LPCTSTR lpszFileSystem
);
```

The **FtpMountStructure** function mounts a different file system or other directory data structure on the server.

Parameters

hClient

Handle to the client session.

lpszFileSystem

A null-terminated string which specifies the file system to mount on the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SMNT command to the server, which may not be supported on some platforms. Use of this command typically requires that the user have administrator privileges on the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCreateDirectory, FtpRemoveDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpNormalizePath Function

```
INT WINAPI FtpNormalizePath(
   HCLIENT hClient,
   LPCTSTR lpszFilePath,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **FtpNormalizePath** function normalizes a file path on the server using the current client connection.

Parameters

hClient

A handle to the client session. This parameter may be INVALID_CLIENT to normalize the path without using the current working directory on the server.

lpszFilePath

A null-terminated string which specifies a file path on the server. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized path when the function returns. The buffer must be large enough to contain the entire path including the terminating null character. If this parameter is NULL it will be ignored and the function will return the length of the normalized path without copying any characters.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the function will return the length of the normalized path without copying any characters.

Return Value

This function returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized path but will not copy any characters. If the function fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will accept a relative or absolute file path on the server and return a normalized version of that path. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. For example, if the current working directory for the user is `/home/jsmith` and the file path provided is:

```
./text/..//data//./files/filename.txt
```

This function would return the following as the normalized path:

```
/home/jsmith/data/files/filename.txt
```

Because the path is relative, the current working directory for the client would be used to create an absolute path. The extraneous path delimiters are removed (the double slashes) and the reference to the parent directory is collapsed to produce a simple, uniform path. This can be useful both for display purposes and to normalize input provided by a user.

If the ***hClient*** parameter is INVALID_CLIENT (in other words, there is no active connection to a server),

relative file paths will not be normalized as absolute paths because there is no current working directory available. This allows the function to be used without an active connection, removing any extraneous path delimiters and collapsing the path without returning an absolute path unless one was provided.

If the path specified by the ***lpszFilePath*** parameter is relative to the current directory and a client handle is not provided, the normalized path returned by the function will always be relative. For example, if the path provided is:

```
../../../../../data/filename.txt
```

This function would return the following as the normalized path:

```
data/filename.txt
```

Because the original path was relative, the normalized path is also relative and will not begin with a leading path delimiter. It is intentionally designed this way to prevent the path from "escaping" to the root directory of a file system, which can present a security risk. If the original path was absolute, the normalized path would also be absolute.

This function will accept an FTP URL and return only the normalized path and file name potion of the URL. If you need to normalize a complete URL, use the **FtpNormalizeUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFilePathUrl, FtpNormalizeUrl, FtpValidateHostName FtpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpNormalizeUrl Function

```
INT WINAPI FtpNormalizeUrl(
   LPCTSTR lpszUrl,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **FtpNormalizeUrl** function normalizes the file path and returns a canonical URL.

Parameters

lpszUrl

A null-terminated string which specifies an FTP URL. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the function returns. The buffer must be large enough to contain the entire URL including the terminating null character. If this parameter is NULL it will be ignored and the function will return the length of the normalized URL without copying any characters.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the function will return the length of the normalized URL without copying any characters.

Return Value

This function returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized URL but will not copy any characters. If the function fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will accept a standard FTP, FTPS or SFTP URL and return a canonical version of that URL. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. If non-standard port numbers are included in the URL, they will be preserved. For example, if the follwing URL is provided to the function:

```
ftps://ftp.example.com:21/file/..//data/filename.txt
```

It will return the following as the normalized URL:

```
ftpes://ftp.example.com/data/filename.txt
```

Because the original URL specified a secure connection on port 21, this is inferred to mean the connection will use explicit TLS and not an implicit connection. This is why the scheme is changed from `ftps://` to `ftpes://` and the port number is removed, since port 21 is the default port for an explicit TLS connection. In addition, the path is collapsed and the extraneous path delimiters are removed.

This function is primarily intended to normalize URLs which may be entered by a user or read from a configuration file. It is not necessary to call this function prior to calling **FtpConnectUrl** because it will always be normalized internally prior to the connection being established. If you wish to check a URL to ensure it is valid and the server name can be resolved to an IP address, use the **FtpValidateUrl** function.

If you need to normalize a remote file path, use the **FtpNormalizePath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnectUrl, FtpGetFilePathUrl, FtpNormalizePath, FtpValidateHostName FtpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpOpenDirectory Function

```
INT WINAPI FtpOpenDirectory(
   HCLIENT hClient,
   LPCTSTR lpszDirectory
);
```

The **FtpOpenDirectory** function opens the specified directory on the server.

Parameters

hClient

Handle to the client session.

lpszDirectory

Pointer to the name of the directory that will be opened. The format of the directory name must match the filename conventions used by the server. If a NULL pointer or an empty string is specified, then the current working directory is opened.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpOpenDirectory** function opens the specified directory on the server using the LIST command. The contents of the directory can be read using the **FtpGetFirstFile** and **FtpGetNextFile** functions. The directory listing is returned on the data channel in one of several different formats. The library can recognize listing formats generated by UNIX, VMS and Windows servers, as well as those of other servers which emulate one of those common formats. Once the complete directory listing has been read, the directory must be closed by calling the **FtpCloseDirectory** function.

Because the directory listing is returned on the data channel, a file transfer cannot be performed while the directory is in the process of being read by the client. Applications which need to collect a list of files to download should first open the directory, read the contents and store the file names in an array. After the directory has been closed, the application can then start transferring the files to the local system.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **FtpChangeDirectory** function and then call this function again, passing NULL or an empty string as the *lpszDirectory* parameter.

To obtain a list of all files in a directory using a single function call, use the **FtpEnumFiles** function. If the server lists files in a format that is not recognized by the library, the **FtpGetFileList** function can be used to obtain an unparsed file listing from the server.

Example

```
if (FtpOpenDirectory(hClient, NULL) != FTP_ERROR)
{
    FTPFILESTATUS ftpFile;
    BOOL bResult;

    bResult = FtpGetFirstFile(hClient, &ftpFile);
    while (bResult)
```

```
    {
        // The ftpFile structure contains information about the file
        bResult = FtpGetNextFile(hClient, &ftpFile);
    }

    FtpCloseDirectory(hClient);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpCloseDirectory, FtpEnumFiles, FtpGetDirectoryFormat, FtpGetFileList, FtpGetFileStatus, FtpGetFirstFile, FtpGetNextFile, FtpSetDirectoryFormat

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpOpenFile Function

```
INT WINAPI FtpOpenFile(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   DWORD dwOpenMode,
   DWORD dwOffset
);
```

The **FtpOpenFile** function creates or opens the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to create or open. The file pathing and name conventions must be that of the server.

dwOpenMode

Specifies the type of access to the file. An application can open a file for reading, create a new file or append data to an existing file. This parameter should be one of the following values.

Value	Description
FTP_FILE_READ	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
FTP_FILE_WRITE	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
FTP_FILE_APPEND	The file is opened for writing on the server. All data will be appended to the end of the file.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error. Typically this indicates that the application failed to call the **FtpCloseFile** function.

It is strongly recommended that most applications use the **FtpGetFile** or **FtpPutFile** functions to perform file transfers. These functions are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

When a file is created on the server, the file ownership and access rights are determined by the server. Some servers may provide a method to change these attributes through site-specific commands. Refer to the server's operating system documentation for more information about what commands may be available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseFile, FtpGetFile, FtpGetFileSize, FtpGetFileTime, FtpPutFile, FtpSetFileTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpOpenFileEx Function

```
INT WINAPI FtpOpenFileEx(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   DWORD dwOpenMode,
   DWORD dwReserved,
   ULARGE_INTEGER uiOffset
);
```

The **FtpOpenFileEx** function creates or opens the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to create or open. The file pathing and name conventions must be that of the server.

dwOpenMode

Specifies the type of access to the file. An application can open a file for reading, create a new file or append data to an existing file. This parameter should be one of the following values.

Value	Description
FTP_FILE_READ	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
FTP_FILE_WRITE	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
FTP_FILE_APPEND	The file is opened for writing on the server. All data will be appended to the end of the file.

dwReserved

An unsigned integer value that is reserved for future use. This parameter should always have a value of zero.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error. Typically this indicates that the application failed to call the **FtpCloseFile** function.

It is strongly recommended that most applications use the **FtpGetFileEx** or **FtpPutFileEx** functions to perform file transfers. These functions are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

When a file is created on the server, the file ownership and access rights are determined by the server. Some servers may provide a method to change these attributes through site-specific commands. Refer to the server's operating system documentation for more information about what commands may be available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCloseFile, FtpGetFileEx, FtpGetFileSizeEx, FtpGetFileTime, FtpPutFileEx, FtpSetFileTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpProxyConnect Function

```
HCLIENT WINAPI FtpProxyConnect(
   UINT nProxyType,
   LPCTSTR lpszProxyHost,
   UINT nProxyPort,
   LPCTSTR lpszProxyUser,
   LPCTSTR lpszProxyPassword,
   LPCTSTR lpszRemoteHost,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **FtpProxyConnect** function establishes a connection through a proxy server.

Parameters

nProxyType

An identifier which specifies the type of proxy server that is being connected to. This value must be defined as one of the following values:

Value	Description
FTP_PROXY_NONE	This value specifies that no proxy server is being used. In this case, the **FtpConnect** function is called directly, ignoring the proxy parameters.
FTP_PROXY_USER	This value specifies that the client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
FTP_PROXY_LOGIN	This value specifies that the client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
FTP_PROXY_OPEN	This value specifies that the client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
FTP_PROXY_SITE	This value specifies that the client is logged into the server. The SITE command is sent, specifying the host name, followed by the username and the password.
FTP_PROXY_OTHER	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the FtpCommand function to send specific command sequences.

lpszProxyHost

The name of the proxy server to connect through; this may be a fully-qualified domain name or an IP address.

nProxyPort

The port number the proxy server is listening on; a value of zero specifies that the default port number should be used.

lpszProxyUser

The user name used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a username is required.

lpszProxyPassword

The password used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a password is required.

lpszRemoteHost

The name of the server that you want to connect to, through the proxy server.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require

	authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_KEEPALIVE	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure TLS connection with the server. This option is typically used when connecting to

	a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller

	files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members ***dwSize***, ***lpszCertStore*** and ***lpszCertName*** must be defined. Undefined structure members must be initialized to a value of zero or NULL and the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **FtpProxyConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The username and password that is used to authenticate the client with the proxy server are not the same as those used to login to the target server. Once a connection has been established with the proxy server, the client must call the **FtpLogin** function to actually login to the server and begin a file transfer.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when

a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnect, FtpCreateSecurityCredentials, FtpDeleteSecurityCredentials, FtpDisconnect, FtpGetSecurityInformation, FtpInitialize, FtpLogin, FtpSetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpPutData Function

```
INT WINAPI FtpPutData(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPVOID lpvBuffer,
   DWORD dwLength,
   DWORD dwReserved
);
```

The **FtpPutData** function transfers the contents of the specified buffer to a file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpvBuffer

The data that will be copied to the server and stored in the specified file.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If the ***lpvBuffer*** parameter is pointing to a Unicode string, it is important to note that the value of the ***dwLength*** parameter should specify the number of bytes, not the number of characters. When using UTF-16, each character is two bytes long and therefore the length of the buffer is effectively double the length of the string. Because Unicode strings can contain null characters, you must also set the current file type to FILE_TYPE_IMAGE prior to calling this function.

The **FtpPutText** function can be used to create a text file from the contents of a string.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetData, FtpGetFile, FtpGetTransferStatus, FtpPutFile, FtpRegisterEvent, FtpSetBufferSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpPutFile Function

```
INT WINAPI FtpPutFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset
);
```

The **FtpPutFile** function transfers the specified file on the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be uploaded. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be created, replaced or appended to on the local system. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetData, FtpGetFile, FtpGetMultipleFiles, FtpGetTransferStatus, FtpPutData, FtpPutFileEx, FtpPutMultipleFiles, FtpRegisterEvent, FtpSetBufferSize, FtpVerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpPutFileEx Function

```
INT WINAPI FtpPutFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset
);
```

The **FtpPutFileEx** function transfers the specified file on the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be uploaded. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be created, replaced or appended to on the local system. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetData, FtpGetFileEx, FtpGetMultipleFiles, FtpGetTransferStatusEx, FtpPutData, FtpPutMultipleFiles, FtpRegisterEvent, FtpSetBufferSize, FtpVerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpPutMultipleFiles Function

```
INT WINAPI FtpPutMultipleFiles(
   HCLIENT hClient,
   LPCTSTR lpszLocalDirectory,
   LPCTSTR lpszRemoteDirectory,
   LPCTSTR lpszFileMask,
   DWORD dwReserved
);
```

The **FtpPutMultipleFiles** function copies one or more files from the local host to the server, using the specified wildcard.

Parameters

hClient

Handle to the client session.

lpszLocalDirectory

Pointer to a null-terminated string which specifies the local directory where the files will be downloaded from. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs. A NULL pointer or empty string specifies that files should be copied to the current working directory.

lpszRemoteDirectory

Pointer to a null-terminated string which specifies the remote directory where the files will be uploaded to. A NULL pointer or empty string specifies that the files should be copied to the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the local system to the server. The file mask should follow the Windows conventions used for wildcard file matches.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpPutMultipleFiles** function is used to transfer files from the local host to the server which match a specified wildcard file mask. All files will be transferred using the current file type as specified by the **FtpSetFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpGetFile, FtpGetMultipleFiles, FtpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpPutText Function

```
INT WINAPI FtpPutText(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   LPCTSTR lpszBuffer
);
```

The **FtpPutText** function creates a text file on the server using the contents of a string buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A null-terminated string that specifies the text file on the server that will be created or overwritten. The file pathing and name conventions must be that of the server.

lpszBuffer

A null-terminated string that contains the text that will be stored in the file.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpPutText** function is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. This function will always set the file type to FILE_TYPE_ASCII before creating the file, and will restore the default file type before the method returns.

If the Unicode version of this function is called, the string will be converted to UTF-8 and then uploaded to the server. If you wish to store the contents of the string as UTF-16 Unicode on the server, you must set the current file type to FILE_TYPE_IMAGE and use the **FtpPutData** function. This function should never be used to upload binary data.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpEnableEvents, FtpGetText, FtpGetTransferStatus, FtpPutData, FtpPutFile, FtpRegisterEvent, FtpSetBufferSize

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRead Function

```
INT WINAPI FtpRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **FtpRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

When **FtpRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpEnableEvents, FtpGetData, FtpGetFile, FtpIsBlocking, FtpIsReadable, FtpIsWritable, FtpRegisterEvent, FtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRegisterEvent Function

```
INT WINAPI FtpRegisterEvent(
   HCLIENT hClient,
   UINT nEventId,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **FtpRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
FTP_EVENT_CONNECT	The control connection to the server has completed.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be

	used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event occurs when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event occurs when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.
FTP_EVENT_QUEUE	This event occurs during a queued file transfer. It only occurs when a file is being transferred using the queue management APIs and the internal state of the queue has changed. The handle passed to this event will be a handle to the queue, not a client connection to the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpRegisterEvent** function associates a callback function with a specific event. The event handler is an **FtpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a file is being uploaded or downloaded. The FTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one FTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpDisableEvents, FtpEnableEvents, FtpEventProc, FtpFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRegisterFileType Function

```
BOOL WINAPI FtpRegisterFileType(
   HCLIENT hClient,
   LPCTSTR lpszExtension,
   UINT nFileType
);
```

The **FtpRegisterFileType** function associates a file name extension with a specific file type.

Parameters

hClient

Handle to the client session. If this parameter is INVALID_CLIENT, the file type is registered for all client sessions in the current process. If the parameter specifies a valid client session, then the association is made only for that specific session.

lpszExtension

A null-terminated string which specifies the file name extension. If this parameter is NULL or points to an empty string, the default file type will be changed for the client session.

nFileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values.

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary image and no data conversion of any type is performed on the file. This is typically the default file type for data file transfers. If the type of file that is being transferred is unknown, this file type should always be used.

Return Value

If the function succeeds, the return value non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpRegisterFileType** function is used to associate specific file types with file name extensions. The library has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that list for the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetAutoFileType, FtpOpenFile, FtpRegisterFileType, FtpSetFileMode, FtpSetFileStructure, FtpSetFileType, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRemoveDirectory Function

```
INT WINAPI FtpRemoveDirectory(
   HCLIENT hClient,
   LPCTSTR lpszDirectory
);
```

The **FtpRemoveDirectory** function removes the specified directory on the server.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the RMD command to create the directory. The user must have the appropriate permission to remove the specified directory. Most servers will not permit you to remove a directory if it contains one or more files.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeDirectory, FtpCreateDirectory, FtpDeleteFile, FtpGetDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRemoveQueuedFile Function

```
BOOL WINAPI FtpRemoveQueuedFile(
    HQUEUE hQueue,
    DWORD dwFileId
);
```

The **FtpRemoveQueuedFile** removes the specified file from the queue.

Parameters

hQueue

A handle to a file transfer queue.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

Files can only be removed from the queue when in an idle or paused state. If you attempt to remove a file while the queue manager is in the process of uploading or download files, the function will fail.

The file identifier can be obtained by several different functions, including **FtpEnumQueuedFiles** and **FtpGetQueueStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpAddQueuedFile, FtpClearQueue, FtpEnumQueuedFiles, FtpGetQueuedFile, FtpGetQueueStatus, FtpSuspendQueue, FtpResetQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRenameFile Function

```
INT WINAPI FtpRenameFile(
   HCLIENT hClient,
   LPCTSTR lpszOldFileName,
   LPCTSTR lpszNewFileName
);
```

The **FtpRenameFile** function renames or moves the specified file on the server. The original file must exist and the current user must have the appropriate permissions to change the file name.

Parameters

hClient

Handle to the client session.

lpszOldFileName

A null-terminated string which specifies the name of the remote file to rename or move. This parameter cannot be NULL or a zero-length string and the file must exist on the server. The file path and name must match the conventions used on the server.

lpszNewFileName

A null-terminated string which specifies the new name for the file. This parameter cannot be NULL or a zero-length string. The file path and name must match the conventions used on the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If the *lpszNewFileName* parameter specifies a file which already exists on the server, the contents of that file will be deleted and replaced by the contents of the file specified by *lpszOldFileName*. If you want to prevent this from occurring, your application can check if a file with the new name exists by calling the **FtpGetFileStatus** function. If successful, you can warn the user the rename operation would delete the contents of the new file as a consequence of renaming the original file.

If the connection was established using the standard FTP or FTPS protocols, this function causes two separate commands to be sent to the server, RNFR and RNTO. If either command fails, the function will fail and return an error code. If the connection was established using SFTP (SSH) the client will request the server rename the file atomically by specifying the internal SSH_FXF_RENAME_ATOMIC option. If the server does not support this option and *lpszNewFileName* already exists on the server, this function will attempt to delete the file and retry the rename operation.

This function can be used to move a file from one folder to another on the server, as long as the user has the appropriate permissions required to perform the operation. If the new file name includes a path, that path must already exist on the server or the function will fail.

If the old and new file names specify folders instead of regular files, the server may or may not allow the operation, depending on the access rights for those folders and the server's configuration. Some servers may allow a simple rename operation within the same parent folder, but may not permit you to move the folder to another location in the filesystem.

There is no guarantee the rename operation will be performed in an atomic fashion if multiple client sessions attempt to rename the same file at the same time. For example, it is possible that one client

could attempt to rename a file while another client is already in the process of moving the file to a new folder. In this case, the server may respond with an error indicating the file cannot be found, an access denied error or a general failure error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpDeleteFile, FtpGetFile, FtpGetFileStatus, FtpGetFileSize, FtpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpReset Function

```
INT WINAPI FtpReset(
   HCLIENT hClient
);
```

The **FtpReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The client cannot be reset while a file transfer is in progress or if the client is in a blocked state. To abort a file transfer, use the **FtpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpResetQueue Function

```
BOOL WINAPI FtpResetQueue(
    HQUEUE hQueue,
    DWORD dwOptions
);
```

The **FtpResetQueue** function resets the state of all file transfers in the current queue.

Parameters

hQueue

A handle to a file transfer queue.

dwOptions

An unsigned integer value which specifies which files should be reset in the queue. It may be one of the following values:

Value	Description
FTP_QUEUE_RESET_ALL	All files in the queue should be reset to their initial state.
FTP_QUEUE_RESET_COMPLETED	All files in the queue which have been successfully transferred will be reset to their initial state.
FTP_QUEUE_RESET_FAILED	All files in the queue which were not transferred will be reset to their initial state.
FTP_QUEUE_RESET_CANCELED	All files in the queue which were canceled will be reset to their initial state.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

When the **FtpStartQueue** function is called multiple times using the same queue handle, it will not attempt to transfer files which have already been successfully copied, and it will not attempt to re-transfer files if the previous transfer failed. This function resets queued files back to their initial state, prior to when the queue manager attempted to perform the transfer. The most common use would be to call the function with the FTP_QUEUE_RESET_FAILED mode and then call **FtpStartQueue** to retry failed or canceled file transfers.

Because canceled file transfers are also considered failed transfers, specifying FTP_QUEUE_RESET_FAILED will reset queued files which either encountered an error during the transfer or were explicitly canceled by calling the **FtpCancelQueuedFile** function. The FTP_QUEUE_RESET_CANCELED option will only reset the state of queued file transfers which were canceled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancelQueuedFile, FtpGetQueuedFile, FtpResumeQueue, FtpStartQueue, FtpStopQueue, FtpSuspendQueue

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpResumeQueue Function

```
BOOL WINAPI FtpResumeQueue(
    HQUEUE hQueue
);
```

The **FtpResumeQueue** function resumes transfers in the queue.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function signals the queue manager to resume processing files in the transfer queue after the **FtpSuspendQueue** function has been called. This function will fail if the queue is in an idle (stopped) state, and will be ignored if the queue manager is already transferring files. The **FtpGetQueueStatus** function can be called to obtain the current status of the queue.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetQueueStatus, FtpResetQueue, FtpStartQueue, FtpStopQueue, FtpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetActivePorts Function

```
INT WINAPI FtpSetActivePorts(
   HCLIENT hClient,
   UINT nLowPort,
   UINT nHighPort
);
```

The **FtpSetActivePorts** function changes the range of local port numbers used for active mode file transfers.

Parameters

hClient

Handle to the client session.

nLowPort

An unsigned integer that specifies the low port number.

nHighPort

An unsigned integer that specifies the high port number.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to modify the range of local port numbers used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **FtpSetActivePorts** function can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

To determine the current range of active port numbers being used, call the **FtpGetActivePorts** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetActivePorts, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetBufferSize Function

```
INT WINAPI FtpSetBufferSize(
   HCLIENT hClient,
   INT nBufferSize
);
```

The **FtpSetBufferSize** function sets the size in bytes of an internal buffer that will be used during data transfers.

Parameters

hClient

Handle to the client session.

nBufferSize

The size of an internal buffer, in bytes. Any value greater than or equal to zero is acceptable. If ***nBufferSize*** is zero, then the default value of 4096 will be used. If ***nBufferSize*** is less than 256 bytes, the buffer size will be set to 256. The maximum value of ***nBufferSize*** is 1048576 (1Mb).

Return Value

If the function succeeds, the return value is the size of the internal buffer that will be used. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a multiple of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetBufferSize, FtpGetData, FtpGetFile, FtpPutData, FtpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetChannelMode Function

```
INT WINAPI FtpSetChannelMode(
    HCLIENT hClient,
    INT nChannel,
    INT nMode
);
```

The **FtpSetChannelMode** function changes the security mode for the specified communication channel.

Parameters

hClient

Handle to the client session.

nChannel

An integer value which specifies which channel to return information for. It may be one of the following values:

Value	Description
FTP_CHANNEL_COMMAND	Change information for the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Change information for the data channel. This is the communication channel used to send or receive data during a file transfer.

nMode

An integer value which specifies the new mode for the specified channel. It may be one of the following values:

Value	Description
FTP_CHANNEL_CLEAR	Data sent and received on this channel should not be encrypted.
FTP_CHANNEL_SECURE	Data sent and received on this channel should be encrypted. Specifying this option requires that a secure connection has already been established with the server.

Return Value

If the function succeeds, the return value is the previous mode for the specified channel. If the function fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetChannelMode** function is used to change the default mode for the specified channel, and is typically used to control whether or not data is encrypted during a file transfer. If a standard, non-secure connection has been established with the server, an error will be returned if you specify the FTP_CHANNEL_SECURE mode for either channel.

If you have established a secure connection and then specify the FTP_CHANNEL_CLEAR mode for the command channel, the client will send the CCC command to the server to indicate that commands

should no longer be encrypted. If the server does not support this command, an error will be returned and the channel mode will remain unchanged. Once the command channel has been changed to clear mode, it cannot be changed back to secure mode. You must disconnect and re-connect to the server if you want to resume sending commands over an encrypted channel.

Changing the mode for the data channel requires that the server support the PROT command. If this command is not supported by the server, the function will fail and the channel mode will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetChannelMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetDefaultZoneId Function

```
BOOL WINAPI FtpSetDefaultZoneId(
    HCLIENT hClient,
    DWORD dwZoneId
);
```

The **FtpSetDefaultZoneId** function sets the default security zone for downloaded files.

Parameters

hClient

Handle to the client session.

dwZoneId

An unsigned integer which specifies the default security zone identifier. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it. This typically indicates that the file was created locally or has not been marked by Windows as originating from another zone. This is the default behavior if the application does not explicitly set a default security zone.
SECURITY_ZONE_LOCAL	The file is considered to be part of the **Local Machine** zone, which includes files stored on the user's local system or removable media. This zone has the least restrictive security settings, and files are typically allowed to run without prompting or limitation. ***This zone should not be assigned to downloaded files***, as doing so can bypass security mechanisms designed to protect against potentially harmful content. Only use this setting for files that are created locally by the application itself and are known to be safe. Misuse of this setting can result in a significant security risk, especially if the file contains scripts, executables, or other active content.
SECURITY_ZONE_INTRANET	The file originates from the **Local Intranet** zone, which includes content served from internal network resources, such as shared drives, file servers, and internal web applications. This zone typically has fewer security restrictions than the Internet zone but more than the Local Machine zone. It is commonly used in enterprise environments where network resources are managed and considered relatively safe. ***This zone should not be used for files downloaded from the public Internet***, as doing so may incorrectly signal that the file is trusted within the organization. Assign this zone only to files that are retrieved from authenticated internal sources or known secure locations on the local network.
SECURITY_ZONE_TRUSTED	The file is from a site or source explicitly added to the

	Trusted Sites zone in Internet Explorer or Windows security settings. This zone is intended for domains or locations that the user or system administrator considers to be safe, such as internal applications hosted on the public Internet or third-party services with a strong reputation for safety. Files marked with this zone typically bypass certain security checks, and applications may allow active content (such as scripts or macros) to run without prompting. Use caution when assigning this zone to downloaded files, as it may override protections that would otherwise apply to untrusted content.
SECURITY_ZONE_INTERNET	The file was downloaded from the **Internet** zone, which represents untrusted or unknown sources outside the local network. This is the standard zone identifier automatically applied by web browsers, email clients, and many applications when a file is downloaded from a public website or external server. Windows uses this zone to enforce security restrictions, such as prompting the user before executing scripts, blocking active content, or displaying a warning when the file is opened. This is typically the appropriate zone to assign for any file obtained over FTP from an external site. Assigning this zone helps ensure that users are notified before opening potentially unsafe content, in accordance with Windows Attachment Manager and Group Policy settings.
SECURITY_ZONE_RESTRICTED	The file is associated with the **Restricted Sites** zone, which enforces the strictest security policies. This zone is used for content that is explicitly marked as untrusted or potentially harmful. Files tagged with this zone may be blocked from execution, prevent active content from running, or trigger warnings when accessed. Assigning this zone can be useful for sandboxing unverified downloads or limiting the risk from files obtained from questionable sources. However, it may also cause legitimate files to fail to open or operate with reduced functionality depending on system policy and application behavior.

Return Value

If the function succeeds, the return value is non-zero. If the function fails it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetDefaultZoneId** function sets the default security zone identifier that will be applied to all files downloaded during the client session. This setting controls how Windows treats downloaded files with respect to security prompts and content execution restrictions. Once the default zone ID is set, it will be automatically applied to all subsequent file downloads performed by the client session. This eliminates the need to manually set the security zone for each individual file, ensuring consistent zone tagging across the session.

When is specified, no alternate data stream is created. This

`SECURITY_ZONE_NONE` `Zone.Identifier`

causes the file to be treated as if it originated from the local system, and Windows will not apply zone-based security restrictions. This is the default behavior which maintains backwards compatibility with previous versions of the library that did not create the security zone metadata for files.

The zone ID metadata is interpreted by Windows components such as **Attachment Execution Services** and **SmartScreen** to determine whether the user should be warned when opening a file. In enterprise environments, Group Policy settings may alter how different zone IDs are handled, such as blocking scripts, macros, or unsigned executables. Developers should test zone assignment behavior in target environments when distributing files to end users.

Note that this function does not retroactively modify existing files. The default zone identifier is applied only at the time a file is downloaded using functions such as **FtpDownloadFile**.

If your application is distributed in a controlled environment, such as a corporate network with endpoint protection or user access controls, consult your organization's security policy before modifying default zone behavior. Improper zone tagging can impact file access, generate security prompts, or interfere with digital signature enforcement.

Be aware that some file operations (such as extracting from a ZIP archive, or copying files to non-NTFS volumes or across certain network shares) may strip the `Zone.Identifier` metadata. Developers should not rely solely on zone tagging as a persistent trust mechanism outside of the initial download context.

To get the security zone identifier for a local file, use the **FtpGetFileZoneInfo** function. To change or remove the security zone information for a file, use the **FtpSetFileZoneInfo** function.

Assigning `SECURITY_ZONE_LOCAL` as the default security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetDefaultZoneId, FtpGetFile, FtpGetFileZoneInfo, FtpSetFileZoneInfo FtpGetMultipleFiles,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetDirectoryFormat Function

```
INT WINAPI FtpSetDirectoryFormat(
    HCLIENT hClient,
    INT nFormatId
);
```

The **FtpSetDirectoryFormat** function is used to specify the format used by the server when returning a list of files. The format type is used internally by the library when parsing the file list returned by the server.

Parameters

hClient

Handle to the client session.

nFormatId

An identifier used to specify the format of the file list returned by the server. The following values are recognized:

Value	Description
FTP_DIRECTORY_AUTO	This value specifies that the library should automatically determine the format of the file lists returned by the server. It is recommended that most applications use this value and allow the library to automatically determine the appropriate file listing format used by the server.
FTP_DIRECTORY_UNIX	This value specifies that the server returns file lists in the format commonly used by UNIX servers. Note that many servers can be configured to return file listings in this format, even if they are not actually a UNIX based platform. Consult the technical reference documentation for your server for more information.
FTP_DIRECTORY_MSDOS	This value specifies that the server returns file lists in the format commonly used by MS-DOS based systems. This includes Windows IIS servers. Long file names will be returned if supported by the underlying filesystem, such as NTFS or FAT32.
FTP_DIRECTORY_VMS	This value specifies that the server returns file lists in the format commonly used by VMS servers. Note that VMS servers can be configured to return a standard UNIX style listing in additional to the default VMS format.
FTP_DIRECTORY_STERLING_1	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 13 byte status code.
FTP_DIRECTORY_STERLING_2	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 10 byte status code.

FTP_DIRECTORY_NETWARE	This value specifies that the server returns file listings in a proprietary format used by Netware servers. The format is similar to UNIX style listings except that file access and permissions are indicated by letter codes enclosed in brackets. This is the default format selected if the server identifies itself as a Netware system.
FTP_DIRECTORY_MLSD	This value specifies that the server should return file listings in a machine-independent format as defined by RFC 3659. This format specifies file information as a sequence of name/value pairs, with the same format being used regardless of the operating system that the server is hosted on. Note that not all servers support this format, and some proxy servers may reject the command even if the remote server supports its use.

Return Value

If the function succeeds, the return value identifies the file list format used by the server. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function should only be used when the library cannot automatically determine the directory format returned by the server. To determine the format used by a server after a file list has been retrieved, use the **FtpGetDirectoryFormat** function.

The default directory format is determined both by the server's operating system and by analyzing the format of the data returned by the server. If the library is unable to automatically determine the format, it will attempt to parse the list of files as though it is a UNIX style listing.

If the FTP_DIRECTORY_MLSD format is specified, the file information returned by the server may differ from the default output of the LIST command. For example, on a UNIX based FTP server, the output of the LIST command is typically the same format that is used by the /bin/ls command, where file names are sorted and hidden files are not listed. However, the MLSD command may return an unsorted list of files that includes hidden files and directories.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCloseDirectory, FtpGetDirectoryFormat, FtpGetFileStatus, FtpGetFirstFile, FtpGetNextFile, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFeatures Function

```
DWORD WINAPI FtpSetFeatures(
    HCLIENT hClient,
    DWORD dwFeatures
);
```

The **FtpSetFeatures** function specifies the server features available to the client.

Parameters

hClient

Handle to the client session.

dwFeatures

An unsigned integer that specifies one or more features. Refer to the documentation for the **FtpGetFeatures** function for a list of available features.

Return Value

If the function succeeds, the return value specifies the features that were previously enabled. If the function fails, it will return zero. Because it is possible that no features were enabled, a return value of zero does not always indicate an error. An application should call **FtpGetLastError** to determine if an error code has been set.

Remarks

The **FtpSetFeatures** function is used to enable a specific set of features for the current session. When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. To enable or disable a specific feature, an application can use the **FtpEnableFeature** function.

For example, the first time an application calls the **FtpGetFileSize** function to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpEnableFeature, FtpGetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFileMode Function

```
INT WINAPI FtpSetFileMode(
   HCLIENT hClient,
   UINT nMode
);
```

The **FtpSetFileMode** function sets the default file transfer mode for the current client session.

Parameters

hClient

Handle to the client session.

nMode

Specifies the default type of data transfer mode for files being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_MODE_STREAM	The data is transmitted as a stream of bytes. This is the default client transfer mode.
FILE_MODE_BLOCK	The data is transmitted as a series of data blocks preceded by one or more header bytes. This transfer mode is currently not supported.
FILE_MODE_COMPRESSED	The data is transmitted in compressed form. This transfer mode is currently not supported.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The file transfer mode should be set before a file is opened or created on the server. Once the transfer mode is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpOpenFile, FtpSetFileStructure, FtpSetFileType, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFileNameEncoding Function

```
INT WINAPI FtpSetFileNameEncoding(
    HCLIENT hClient,
    INT nEncoding
);
```

The **FtpSetFileNameEncoding** function specifies what type of encoding will be used when file names are sent to the server.

Parameters

hClient

Handle to the client session.

nEncoding

An integer value which specifies the encoding type. It may be one of the following values:

Value	Description
FTP_ENCODING_ANSI	File names are sent as 8-bit characters using the default character encoding for the current codepage. If the Unicode version of the functions are used, file names are converted from Unicode to ANSI using the current codepage before being sent to the server. This is the default encoding type.
FTP_ENCODING_UTF8	File names that contain non-ASCII characters are sent using UTF-8 encoding. This encoding type is only available on servers that advertise support for UTF-8 encoding and permit that encoding type to be enabled by the client.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetFileNameEncoding** function can be used to enable UTF-8 encoding of file names, which provides improved support for the use of international character sets. However, the server must provide support for UTF-8 encoding by advertising it in response to the FEAT command and it must support the OPTS command which is used to enable UTF-8 encoding. If the server does not advertise support for UTF-8, or the OPTS command fails with an error, then this function will fail with an error and the encoding type will not change.

Although it is possible to use the **FtpEnableFeature** function to explicitly enable the FTP_FEATURE_UTF8 feature, this is not recommended. If the server has not advertised support for UTF-8 encoding in response to the FEAT command, that typically indicates that UTF-8 encoding is not supported. Attempting to force UTF-8 encoding can result in unpredictable behavior when file names contain non-ASCII characters.

It is important to note that not all FTP servers support UTF-8 encoding, and in some cases servers which advertise support for UTF-8 encoding do not implement the feature correctly. For example, a server may allow a client to enable UTF-8 encoding, but once enabled will not permit the client to disable it. Some servers may advertise support for UTF-8 encoding, however if the underlying file system does not support UTF-8 encoded file names, any attempt to upload or download a file may fail

with an error indicating that the file cannot be found or created.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCommand, FtpEnableFeature FtpGetFeatures, FtpGetFileNameEncoding, FtpSetFeatures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFilePermissions Function

```
INT WINAPI FtpSetFilePermissions(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   DWORD dwPermissions
);
```

The **FtpSetFilePermissions** function returns information about the access permissions for a specific file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string which contains the name of the file to be updated. The filename cannot contain any wildcard characters.

dwPermissions

An unsigned integer which will specify the new access permissions for the file. The file permissions are represented as bit flags, and may be one or more of the following values combined with a bitwise Or operator:

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.

FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the function succeeds, the return value is a result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the SITE CHMOD command to set the permissions for the file. This command is typically only supported on servers that are hosted on UNIX based systems. If the command is not supported, an error will be returned.

Users who are familiar with the UNIX operating system will recognize the **chmod** command used to change the file permissions. However, it should be noted that the numeric value used as an argument to the command is in octal, not decimal. For example, issuing the command **chmod 644 filename.txt** on a UNIX based system will make the file readable and writable by the owner, and readable by other users in the owner's group as well as all other users. The value 644 is an octal value, which is equivalent to the decimal value 420. If you were to mistakenly specify 644 as the value for the *dwPermissions* parameter, rather than the decimal value of 420, the permissions on the file would be incorrect. It is strongly recommended that you use the pre-defined constants to prevent this sort of error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFilePermissions, FtpGetFileStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFileStructure Function

```
INT WINAPI FtpSetFileStructure(
    HCLIENT hClient,
    UINT nType
);
```

The **FtpSetFileStructure** function sets the default file structure for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

hClient

Handle to the client session.

nType

Specifies the default type of file structure being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_STRUCT_NONE	The file has no inherent structure and is considered to be a stream of bytes. This is the default structure for file transfers.
FILE_STRUCT_RECORD	The file uses a record structure. This file structure is currently not supported.
FILE_STRUCT_PAGE	The file uses a page structure. This file structure is currently not supported.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The file structure should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpOpenFile, FtpSetFileMode, FtpSetFileType, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFileTime Function

```
INT WINAPI FtpSetFileTime(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPSYSTEMTIME lpFileTime
);
```

The **FtpSetFileTime** function sets the modification time for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a SYSTEMTIME structure that specifies the new modification time for the remote file.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetFileTime** function will change the modification time of a file on the server. The values specified in the SYSTEMTIME structure are expected to represent UTC time, not time adjusted for the local system's timezone. If the values do represent the local time, it must be converted to UTC time prior to calling this function. To populate the SYSTEMTIME structure with the current time, use the **GetSystemTime** function.

When connected to an FTP server, this function uses the MDTM command to set the modification time for the specified file. Not all servers implement this command, in which case the function call will fail. Note that some servers only support the MDTM command to return, but not change, the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFileStatus, FtpGetFileTime, FtpOpenDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFileType Function

```
INT WINAPI FtpSetFileType(
   HCLIENT hClient,
   UINT nFileType
);
```

The **FtpSetFileType** function sets the default file type for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

hClient

Handle to the client session.

nFileType

Specifies the default type of file being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the **FtpRegisterFileType** function.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value

is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The file type should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created. Some functions, such as **FtpOpenDirectory** and **FtpGetText**, will temporarily change the default file type to FILE_TYPE_ASCII and then restore the current file type when they return.

Calling this function has no practical effect when connected to an SFTP (SSH) server. They do not differentiate between text and binary files and the default file type will always be FILE_TYPE_IMAGE. If your application is uploading or downloading a text file, this difference between FTP and SFTP is important because the operating system that hosts the server may have different end-of-line character conventions than the client system. For example, if you download a text file from a UNIX system using SFTP, the end-of-line is indicated by a single linefeed (LF) character. However, on the Windows platform, the end-of-line is indicated by a carriage-return and linefeed sequence (CRLF).

The **FtpGetFileType** function can be used to determine the current file type. To determine the transfer file type for a specific file, use the **FtpGetAutoFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetAutoFileType, FtpGetFileType, FtpOpenFile, FtpRegisterFileType, FtpSetFileMode, FtpSetFileStructure, FtpSetPassiveMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetFileZoneInfo Function

```
BOOL WINAPI FtpSetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   DWORD dwZoneId,
   LPCTSTR lpszHostUrl
);
```

The **FtpSetFileZoneInfo** function sets or clears the zone identifier information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and write to the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

dwZoneId

An unsigned integer which specifies the zone ID which should be assigned to the file. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string which specifies the URL used to originally download the file. If this information is not required the parameter may be a NULL pointer or an empty string. If an originating URL is provided, it must be a valid FTP URL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetFileZoneInfo** function enables applications to set or remove the security zone identifier metadata associated with a local file. This information is stored in an alternate data stream named `:Zone.Identifier` and is used by Windows to determine the origin and trust level of a file, potentially prompting the user with a warning before the file is opened.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This

includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be applied to files on network volumes because the SMB protocol does not support alternate data streams. If a file with zone metadata is copied from a local NTFS drive to a network share, any associated streams like `Zone.Identifier` will be lost.

Specifying `SECURITY_ZONE_NONE` will remove any existing zone identifier stream from the file. This has the same effect as selecting the Unblock option in the file's properties dialog in Windows Explorer. Once the stream is removed, the file will be treated as though it originated from the local computer. Note that when the `Zone.Identifier` stream is deleted from a file to remove its security zone information, Windows Explorer may not immediately reflect this change. This is because Explorer caches certain file metadata, including zone information used to display the "This file came from another computer" warning in the file's Properties dialog.

When assigning a zone ID, it is important to ensure that the value accurately reflects the source of the file. For files that have been downloaded from the Internet, it is strongly recommended to use `SECURITY_ZONE_INTERNET`. This allows Windows and other software to enforce appropriate security measures, such as requiring user confirmation before executing or opening the file.

In most scenarios, zone identifiers should only be set for files that were downloaded from untrusted or external sources. Setting an overly permissive zone (such as `SECURITY_ZONE_LOCAL`) for files obtained from the Internet or other potentially unsafe locations can expose users to security risks and circumvent built-in protections.

If a URL is provided via the *lpszHostUrl* parameter, it will be included in the zone metadata and can be retrieved later using the **FtpGetFileZoneInfo** function. This provides additional context about the origin of the file and may aid in auditing or diagnostic tasks. The URL must be a syntactically valid FTP address and should reflect the actual location from which the file was downloaded.

Not all file systems support alternate data streams. The function will fail if used on a volume (such as FAT32) that does not support this feature. Additionally, modifying zone information may fail if the file is in use, read-only, or if the current process does not have the required permissions. It is possible to disable the creation of security zone metadata through local policy settings. In this case, the function will fail and the last error code will be set to `ST_ERROR_POLICY_NO_ZONEINFO`.

For more detailed information about the security zone identifiers, refer to the **FtpSetDefaultZoneId** function.

Assigning `SECURITY_ZONE_LOCAL` as the security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetDefaultZoneId, FtpGetFile, FtpGetFileZoneInfo, FtpGetMultipleFiles, FtpSetDefaultZoneId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetIdleTimeout Function

```
INT WINAPI FtpSetIdleTimeout(
    HCLIENT hClient,
    INT nIdleTime
);
```

The **FtpSetIdleTimeout** function sets the server-side idle timeout period for the control connection.

Parameters

hClient

Handle to the client session.

nIdleTime

The number of seconds until the control session is considered inactive. A value of zero specifies the default idle timeout period of 300 seconds (5 minutes) will be used. If the value is less than 30 seconds or greater than 7200 seconds, it will be automatically adjusted to fit within the valid range.

Return Value

If the function succeeds, the return value is the previous idle timeout period. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetIdleTimeout** function changes the idle timeout period for the control connection, which can be useful in scenarios where a file transfer session is expected to remain idle for extended periods between commands. For example, when transferring large batches of files or performing periodic synchronization, the server may automatically close the connection if no activity is detected within its default timeout period. Increasing the idle timeout helps prevent unexpected disconnects during these idle periods, reducing the need to repeatedly reconnect and re-authenticate. Conversely, reducing the timeout can be used to enforce shorter sessions in security-sensitive environments.

The `SITE IDLE` command is a non-standard extension to the FTP protocol that allows a client to query or set the server's idle timeout period for the current session. When supported, it enables the client to request how long the server should wait during periods of inactivity before automatically closing the connection. This setting does not affect the duration of data transfers, only how long the control session is kept open when idle.

Because `SITE IDLE` is not part of the official FTP specification, support for it varies between servers. If the command is not supported, the function will fail but the session will remain connected using the server's default timeout settings. The function will also return an error if a negative timeout value is specified or if the session is using SFTP, which does not support this operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCommand, FtpGetIdleTimeout, FtpGetTimeout, FtpSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetLastError Function

```
VOID WINAPI FtpSetLastError(
   DWORD dwErrorCode
);
```

The **FtpSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those functions which call **FtpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **FtpGetLastError** function. The use of **FtpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetErrorString, FtpGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetPassiveMode Function

```
INT WINAPI FtpSetPassiveMode(
   HCLIENT hClient,
   BOOL bPassiveMode
);
```

The **FtpSetPassiveMode** function enables or disables passive mode file transfers for the specified client session.

Parameters

hClient

Handle to the client session.

bPassiveMode

A boolean flag which specifies that the client should enter passive mode and establish all connections with the server to transfer data.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

By default, the File Transfer Protocol uses active mode transfers, whereby the data connection is established from the server back to the local client. However, this can introduce problems for a client application that is behind a proxy server, firewall or a router which uses Network Address Translation (NAT). Enabling passive mode transfers instructs the client to create an outbound connection from the local system to the server for the data connection, similarly to how the control connection is established.

Not all servers may support passive mode, in which case an error will be returned to the client when this function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpGetActivePorts, FtpGetData, FtpGetFile, FtpProxyConnect, FtpPutData, FtpPutFile, FtpSetActivePorts

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetPriority Function

```
INT WINAPI FtpSetPriority(
   HCLIENT hClient,
   INT nPriority
);
```

The **FtpSetPriority** function specifies the priority for file transfers.

Parameters

hClient

Handle to the client session.

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Value	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the function succeeds, the return value is the previous file transfer priority. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetPriority** function can be used to control the processor usage, memory and network

bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the FTP_PRIORITY_BACKGROUND value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The FTP_PRIORITY_CRITICAL priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to FTP_PRIORITY_CRITICAL, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetTimeout Function

```
INT WINAPI FtpSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **FtpSetTimeout** function sets the number of seconds the client will wait for a response from the server.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete. If this value is zero, the default timeout period of 20 seconds will be used.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking function will fail and return to the caller. The timeout value affects both commands issued on the control channel and file transfers using the data channel.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpGetTimeout, FtpIsReadable, FtpIsWritable, FtpProxyConnect, FtpRead, FtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpStartQueue Function

```
BOOL WINAPI FtpStartQueue(
   HQUEUE hQueue,
   DWORD dwQueueMode,
   DWORD dwReserved,
   FTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **FtpStartQueue** function begins transferring files in the queue.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An unsigned integer which specifies which files in the queue should be transferred. It may be one of the following values:

Constant	Description
FTP_QUEUE_ALL	All files in the queue should be transferred.
FTP_QUEUE_DOWNLOAD	Transfer only those files which have been queued for download.
FTP_QUEUE_UPLOAD	Transfer only those files which have been queued for upload.

dwReserved

An unsigned integer reserved for internal use. This parameter value should always be zero.

lpEventProc

Specifies the procedure-instance address of an application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, there will be no callback notifications.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

Queued file transfers are performed asynchronously using a background worker thread. If you provide the address to an event callback function, that function will always be invoked in the context of the queue manager thread. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself and you should avoid calling any functions which could cause the thread to block. For example, you should not attempt to establish other network

connections from within the event handler.

If your application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function. Controls should only be modified by the same UI thread that created their window. One common approach to address this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

To obtain the handle to the queue from within an event handler, call the **FtpGetThreadQueue** function with the *dwThreadId* parameter set to a value of zero. This will return the handle for the current queue management session which invoked the callback function.

If a callback function is provided, the FTP_QUEUE_EVENT event notification will be invoked whenever the internal state of the queue has changed. The handle value passed to the callback function will be a handle to the queue, not to a client connection. The **FtpGetQueueStatus** function can be called within the event handler to obtain the current state of the queue. This will event will always occur when the queue manager begins processing files in the queue and after it has stopped. This event will also occur whenever the queue is suspended and then resumed.

The **FtpWaitForQueue** function can be used to wait for the queue manager to complete processing the transfer queue.

Example

```
// Create a new queue
HQUEUE hQueue = FtpCreateQueue(INFINITE, FTP_TIMEOUT, FTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = FtpAddQueuedFile(
                     hQueue,
                     lpszLocalFile,
                     lpszRemoteFile,
                     FTP_OPTION_DEFAULT,
                     FILE_TYPE_AUTO,
                     FTP_TIMEOUT);

// Display information about the queued file
if (dwFileId != 0)
{
    FTPQUEUEDFILE queuedFile;

    if (FtpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case FTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case FTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
```

```
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
            }
        }
    }

    // Start the queued file transfers and wait for it to complete
    if (FtpStartQueue(hQueue, FTP_QUEUE_ALL, 0, NULL, 0))
    {
        FtpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
        FtpStopQueue(hQueue);
    }

    // Remove all files from the queue
    FtpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpEventProc, FtpGetQueueStatus, FtpGetThreadQueue, FtpResumeQueue, FtpStartQueue, FtpStopQueue, FtpSuspendQueue, FtpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpStopQueue Function

```
BOOL WINAPI FtpStopQueue(
    HQUEUE hQueue
);
```

The **FtpStopQueue** function stops transferring queued files.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

When the **FtpStopQueue** function is called and there is a file transfer in progress, it will not immediately stop the upload or download of the file. Instead, the queue manager is signaled to stop processing additional files in the queue after the transfer has completed. To wait for the current transfer to complete, call the **FtpWaitForQueue** function.

It is permitted to call **FtpStopQueue** from within a queue event handler. In this case, the current file transfer will complete and the queue manager will terminate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancelQueuedFile, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpSuspendQueue, FtpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSuspendQueue Function

```
BOOL WINAPI FtpSuspendQueue(
    HQUEUE hQueue
);
```

The **FtpSuspendQueue** function pauses all file transfers for the specified queue.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

If there is a file transfer in progress when this function is called, the transfer will complete normally. The queue manager will enter a suspended state after the transfer has completed and before it begins processing the next file in the queue.

It is permitted to call **FtpSuspendQueue** from within a queue event handler. In this case, the current file transfer will complete and the queue manager will stop processing additional files in the queue until it is resumed or stopped.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpCancelQueuedFile, FtpGetQueueStatus, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpTaskAbort Function

```
BOOL WINAPI FtpTaskAbort(
   UINT nTaskId,
   DWORD dwMilliseconds
);
```

Abort the specified asynchronous task.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to abort.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskAbort** function signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the *dwMilliseconds* parameter has a value of zero, the function returns immediately after the background thread has been signaled. If the *dwMilliseconds* parameter is non-zero, the function will wait that amount of time for the background thread to terminate.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock. To abort a file transfer within an event handler, use the **FtpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskDone, FtpTaskResume, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpTaskDone Function

```
BOOL WINAPI FtpTaskDone(
    UINT nTaskId
);
```

Determine if an asynchronous task has completed.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed, this function returns a non-zero value. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskDone** function is used to determine if the specified asynchronous task has completed. If you use this function to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this function rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **FtpGetTaskError** function will return the last error code associated with the task. A return value of zero indicates success, while a non-zero return value specifies an error code that indicates the cause of the failure. The last error code for the task can also be retrieved using the **FtpTaskWait** function, which causes the application to wait for the asynchronous task to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetTaskError, FtpTaskAbort, FtpTaskResume, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpTaskResume Function

```
BOOL WINAPI FtpTaskResume(
    UINT nTaskId
);
```

Resume execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskResume** function resumes execution of the background worker thread that was previously suspended using the **FtpTaskSuspend** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskAbort, FtpTaskDone, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpTaskSuspend Function

```
BOOL WINAPI FtpTaskSuspend(
    UINT nTaskId
);
```

Suspend execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskSuspend** function will suspend execution of the background worker thread associated with the task. Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **FtpTaskResume** function. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskAbort, FtpTaskDone, FtpTaskResume, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpTaskWait Function

```
BOOL WINAPI FtpTaskWait(
   UINT nTaskId,
   DWORD dwMilliseconds,
   DWORD dwReserved,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);
```

Wait for an asynchronous task to complete.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to complete.

dwReserved

An unsigned integer reserved for future use. This value should always be zero.

lpdwElapsed

A pointer to an unsigned integer that will contain elapsed time in milliseconds when the function returns. If this information is not required, this parameter may be NULL.

lpdwError

A pointer to an unsigned integer that will contain the error code associated with the completed task. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskWait** function waits for the specified task to complete. If the task is active and the ***dwMilliseconds*** parameter is non-zero, this function will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the ***dwMilliseconds*** parameter is zero, then this function will poll the status of the task and return immediately to the caller.

If the specified task has already completed at the time this function is called, the function will return immediately without causing the current thread to block. If the ***lpdwElapsed*** parameter is not NULL, it will contain the number of milliseconds that it took for the task to complete. If the ***lpdwError*** parameter is not NULL, it will contain the last error code value that was set by the worker thread before it terminated. If this value is zero, that means that the background operation was successful and no error occurred. A non-zero value will indicate that the background operation has failed.

You should not call this function from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this function is blocked waiting for the background task to complete, and this can cause your application to appear non-

responsive to the end-user. If you have a GUI application and you need to periodically check to see if a task has completed, create a timer to periodically call the **FtpTaskDone** function. When it returns a non-zero value (indicating that the task has completed), you can safely call **FtpTaskWait** to obtain the elapsed time and last error code without blocking the current thread.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Example

```
UINT nTaskId;

// Begin a file transfer in the background

nTaskId = FtpAsyncGetFile(hClient,
                          lpszLocalFile,
                          lpszRemoteFile,
                          FTP_TRANSFER_DEFAULT,
                          0,
                          NULL,
                          0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    FtpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpTaskDone, FtpTaskResume, FtpTaskSuspend, FtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpUninitialize Function

```
VOID WINAPI FtpUninitialize();
```

The **FtpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **FtpInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **FtpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **FtpUninitialize** for every successful call to **FtpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpConnect, FtpDisconnect, FtpInitialize, FtpProxyConnect, FtpReset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpUploadFile Function

```
BOOL WINAPI FtpUploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPFTPTRANSFERSTATUS lpStatus
   FTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **FtpUploadFile** function uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A null-terminated string which specifies the file on the local system that will be uploaded. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_DEFAULT	Default options should be used for file transfers. This is the same as specifying that all file transfers should use passive mode when establishing a data connection with the server. It is recommended most applications use passive mode to prevent potential compatibility issues with certain types of firewalls and routers which use Network Address Translation (NAT).
FTP_OPTION_PASSIVE	This option specifies file transfers should attempt to establish the data connection with the server. If the local system is behind a firewall or a NAT router, the server may not be able to establish a data connection back to the client and the transfer will fail. This option forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.

FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpUploadFile** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then upload the file to the server. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be uploaded as a text or binary file. A value of "a" specifies that the file should be uploaded as a text file. A value of "i" specifies that the file should be uploaded as a binary file. If the type is not explicitly specified, the file will be uploaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpUploadFile** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpPutFile** to upload the file.

Example

```
FTPTRANSFERSTATUS ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile =
_T("ftp://update:secret@ftp.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = FtpUploadFile(lpszLocalFile,
                        lpszRemoteFile,
                        FTP_OPTION_PASSIVE,
                        &ftpStatus,
                        NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, FtpDownloadFile, FtpGetTransferStatus, FtpPutFile, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpUploadFileEx Function

```
BOOL WINAPI FtpUploadFileEx(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPFTPTRANSFERSTATUSEX lpStatus
   FTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **FtpUploadFileEx** function uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an

	unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpUploadFileEx** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then upload the file to the server. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] hostname [:port] / [path / ...]
[filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be uploaded as a text or binary file. A value of "a" specifies that the file should be uploaded as a text file. A value of "i" specifies that the file should be uploaded as a binary file. If the type is not explicitly specified, the file will be uploaded as a binary file.

The ***lpStatus*** parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related

to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpUploadFileEx** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpPutFileEx** to upload the file.

Example

```
FTPTRANSFERSTATUSEX ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile =
_T("ftp://update:secret@ftp.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = FtpUploadFileEx(lpszLocalFile,
                          lpszRemoteFile,
                          FTP_OPTION_PASSIVE,
                          &ftpStatus,
                          NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEventProc, FtpDownloadFileEx, FtpGetTransferStatusEx, FtpPutFileEx, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpValidateHostName Function

```
BOOL WINAPI FtpValidateHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **FtpValidateHostName** function determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The function will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the ***nMaxLength*** parameter is zero, it will be ignored and the IP address will not be returned.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the ***lpszAddress*** string buffer. The buffer must be large enough to store the complete address. Because this function can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the ***lpszAddress*** parameter will be ignored.

Return Value

If the function succeeds, the host name is valid and the return value will be non-zero. If the function fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpValidateHostName** function provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. If the Unicode version of this function is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

If the ***lpszHostName*** parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address.

If you wish to validate a complete FTP URL instead of a host name, use the **FtpValidateUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnectUrl, FtpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpValidateUrl Function

```
BOOL WINAPI FtpValidateUrl(
    LPCTSTR lpszUrl
);
```

The **FtpValidateUrl** function determines if a string represents a valid FTP URL.

Parameters

lpszUrl

A null-terminated string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpValidateUrl** function will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure FTP scheme. This function will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this function will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **FtpConnectUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpConnectUrl, FtpNormalizePath, FtpNormalizeUrl, FtpValidateHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpVerifyFile Function

```
BOOL WINAPI FtpVerifyFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
);
```

The **FtpVerifyFile** function attempts to verify that the contents of a file on the local system are the same as the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string which specifies the file on the local system the remote file will be validated against. The file pathing and name conventions should follow Windows conventions. It is permitted to include environment variables in the local path by surrounding the variable with percent (%) symbols. Local file paths may also use standard file URLs.

lpszRemoteFile

A null-terminated string which specifies the file on the server that will be validated. The file naming conventions must be that of the host operating system. It is permitted to use a URL as the remote file name as long as the server name and port number match what was used to establish the connection.

dwOptions

Specifies the options that may be used when comparing the files. This parameter may be one or more of the following values:

Value	Description
FTP_VERIFY_DEFAULT	File verification should use the best option available based on the available server features. If the server supports the XSHA1 or XMD5 command, the function will calculate an SHA-1 or MD5 hash of the local file contents and compare the value with the file on the server. If the server does not support either hash algorithm, but it does support the XCRC command, the function will calculate a CRC32 checksum of the local file contents and compare the value with the file on the server. If the server does not support any hash or checksum commands, the function will compare the size of the local and remote files.
FTP_VERIFY_SIZE	Files are verified by comparing the number of bytes of data in the local and remote files. This is the least reliable option, and should only be used if the server does not support either the XSHA1, XMD5 or XCRC commands.
FTP_VERIFY_CRC32	Files are verified by calculating a CRC32 checksum of the local file contents and comparing it with the value returned by the server in response to the XCRC command. This option should only be used if the server does not support the XSHA1 or XMD5 commands.

FTP_VERIFY_MD5	Files are verified by calculating an MD5 hash of the local file contents and comparing it with the value returned by the server in response to the XMD5 command. This option should only be used if the server does not support the XSHA1 command.
FTP_VERIFY_SHA1	Files are verified by calculating an SHA-1 hash of the local file contents and comparing it with the value returned by the server in response to the XSHA1 command. This is the preferred option for performing file verification.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpVerifyFile** function will attempt to verify that the contents of the local and remote files are identical using one of several methods, based on the features that the server supports. Preference will be given to the most reliable method available, using either an SHA-1 hash, MD5 hash, a CRC32 checksum or comparing the size of the file, in that order.

The method used to verify a file's contents determines how confidently it can be said that the local and remote files are identical. This function supports several algorithms, listed below in order of preference based on reliability and availability:

- SHA-1 is a cryptographic hash function that produces a 160-bit (20-byte) hash value. It is the most reliable option supported by this function and is preferred for verifying file contents. Although no longer recommended for cryptographic security due to potential collision attacks, it remains well-suited for file integrity checks where the threat of intentional tampering is low.
- MD5 generates a 128-bit (16-byte) hash and is faster than SHA-1. It offers a reasonable balance between performance and reliability for non-security-critical applications such as file verification. However, it is less resistant to collisions and should be used only if SHA-1 is not supported by the server.
- CRC32 is a non-cryptographic checksum algorithm that produces a 32-bit value. It is designed for detecting accidental data corruption rather than verifying content integrity. CRC32 is significantly faster but also far more prone to collisions, and should only be used when stronger methods like SHA-1 or MD5 are unavailable.
- If none of the supported hash or checksum commands are available, the function falls back to comparing the local and remote file sizes. This is the least reliable method and can only detect differences in file length, not in actual content.

It is not recommended that you use this function with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **FtpVerifyFile** function to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetFile, FtpGetFileSize, FtpGetFileStatus, FtpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpWaitForQueue Function

```
BOOL WINAPI FtpWaitForQueue(
   HQUEUE hQueue,
   DWORD dwMilliseconds,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);
```

The **FtpWaitForQueue** function waits for the specified queue to complete the file transfers.

Parameters

hQueue

A handle to a file transfer queue.

dwMilliseconds

An unsigned integer value which specifies the number of milliseconds to wait for the queue to complete processing. If this value is zero, the function will return immediately. If this value is INFINITE (0xFFFFFFFF), the function will block indefinitely until all files in the queue have been processed.

lpdwElapsed

A pointer to an unsigned integer which will contain the number of milliseconds the queue manager has been active when the function returns. If this information is not needed, this parameter can be NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

lpdwError

A pointer to an unsigned integer which will contain the last error code set if a file transfer has failed or was canceled. If this information is not needed, this parameter can be NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

Return Value

If the function succeeds, the return value is non-zero, which means the queue manager has completed transferring all queued files. If the function fails, or the timeout period elapses, it will return zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until either the queue manager has completed processing all files in the queue, or until the specified number of milliseconds have elapsed. It is important to note that Windows messages will not be processed during this time. If you call this function within the main UI thread, it can potentially cause the application to become non-responsive. To determine the current state of the queue without blocking the current thread, call the **FtpGetQueueStatus** function.

If the *dwMilliseconds* parameter is non-zero and file transfers have not completed within the specified amount of time, the function will return zero and the last error code will be set to `ST_ERROR_OPERATION_TIMEOUT`.

If an event callback function has been specified when starting the queue, you should never call this function within the event handler. Attempting to do so can potentially create a situation which will cause the application to become non-responsive. The function will return zero if it determines it's being called within the context of the queue manager thread and will set the last error code to `ST_ERROR_THREAD_DEADLOCK`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetQueueStatus, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpStopQueue, FtpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpWrite Function

```
INT WINAPI FtpWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **FtpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpEnableEvents, FtpIsBlocking, FtpIsReadable, FtpIsWritable, FtpPutData, FtpPutFile, FtpRead, FtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Data Structures

- FTPCLIENTQUOTA
- FTPFILESTATUS
- FTPFILESTATUSEX
- FTPQUEUEDFILE
- FTPQUEUESTATUS
- FTPTRANSFERSTATUS
- FTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPCLIENTQUOTA

The **FTPCLIENTQUOTA** structure provides information about the file quota for the current client session.

```
typedef struct _FTPCLIENTQUOTA
{
    DWORD   dwFileCount;
    DWORD   dwFileLimit;
    DWORD   dwDiskUsage;
    DWORD   dwDiskLimit;
} FTPCLIENTQUOTA, *LPFTPCLIENTQUOTA;
```

Members

dwFileCount

An unsigned integer value which specifies the number of files that the user has created. If file quotas have not been enabled for the current user, this value will be zero.

dwFileLimit

An unsigned integer value which specifies the maximum number of files that may be created by the user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskUsage

An unsigned integer value which specifies the number of bytes of disk storage that has been allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskLimit

An unsigned integer value which specifies the maximum number of bytes of disk storage that may be allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPFILESTATUS

This structure is used by the **FtpEnumFiles**, **FtpGetFirstFile** and **FtpGetNextFile** functions to return information about files on the server.

```
typedef struct _FTPFILESTATUS
{
    TCHAR        szFileName[FTP_MAXFILENAMELEN];
    TCHAR        szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR        szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL         bIsDirectory;
    DWORD        dwFileSize;
    DWORD        dwFileLinks;
    DWORD        dwFileVersion;
    DWORD        dwFilePerms;
    SYSTEMTIME   stFileDate;
} FTPFILESTATUS, *LPFTPFILESTATUS;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

dwFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify the

individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the chmod command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

stFileDate

A SYSTEMTIME structure which specifies the date that the file was created or last modified.

File Permissions

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value

A	0	1h
B	1	2h
C	2	4h
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPFILESTATUSEX

This structure is used by the **FtpEnumFilesEx**, **FtpGetFirstFileEx** and **FtpGetNextFileEx** functions to return information about files on the server. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPFILESTATUSEX
{
    TCHAR          szFileName[FTP_MAXFILENAMELEN];
    TCHAR          szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR          szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL           bIsDirectory;
    ULARGE_INTEGER uiFileSize;
    DWORD          dwFileLinks;
    DWORD          dwFileVersion;
    DWORD          dwFilePerms;
    DWORD          dwFileFlags;
    SYSTEMTIME     stFileDate;
} FTPFILESTATUSEX, *LPFTPFILESTATUSEX;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

uiFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the chmod command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

dwFileFlags

This structure member is reserved for future use.

stFileDate

A SYSTEMTIME structure which specifies the date that the file was created or last modified.

File Permissions

Value	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	0	1h
B	1	2h
C	2	4h
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPQUEUEDFILE

This structure is used by the **FtpGetFirstQueuedFile**, **FtpGetNextQueuedFile** and **FtpGetQueuedFile** functions to return information about a file in the transfer queue.

```
typedef struct _FTPQUEUEDFILE
{
    DWORD          dwFileId;
    DWORD          dwQueueMode;
    DWORD          dwQueueFlags;
    DWORD          dwTimeElapsed;
    DWORD          dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
    TCHAR          szRemoteName[FTP_MAXURLPATHLEN];
} FTPQUEUEDFILE, *LPFTPQUEUEDFILE;
```

Members

dwFileId

An unsigned integer which specifies a unique identifier for the queued file. The application should not make any assumptions about the value of the file identifiers. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

dwQueueMode

An unsigned integer value which specifies how the file was queued for transfer. It may be one of the following values:

Constant	Description
FTP_QUEUE_DOWNLOAD	The file was queued for download.
FTP_QUEUE_UPLOAD	The file was queued for upload.

dwQueueFlags

An unsigned integer which specifies one or more bit flags which provides information about the status of the file transfer. It may be one or more of the following values:

Constant	Description
FTP_QUEUE_FLAG_NONE	The file is pending transfer in queue.
FTP_QUEUE_FLAG_COMPLETED	The file has been transferred successfully. If this flag is set, no errors were encountered during the upload or download.
FTP_QUEUE_FLAG_FAILED	The file transfer failed. If this flag is set, the *dwError* member of this structure will contain the error code associated with the failed transfer.
FTP_QUEUE_FLAG_CANCELED	The file transfer was canceled. This flag is only set when the **FtpCancelQueuedFile** function has been called and a queued file is in the process of being uploaded or downloaded.

dwTimeElapsed

An unsigned integer which specifies the number of milliseconds required to complete the transfer. This value will be zero unless the file has been transferred successfully.

dwLastError

An unsigned integer which specifies the last error code for a failed transfer. If the file was transferred successfully, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the file transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

szLocalFile

A null-terminated string which specifies the full path to the local file being transferred.

szRemoteFile

A null-terminated string which specifies the complete URL to the remote file being transferred.

Remarks

It is possible that the ***szLocalFile*** and ***szRemoteFile*** structure members will be different than the values passed to the **FtpAddQueuedFile** function. Those values are normalized, with any relative paths converted to absolute paths. Internationalized domain names will be encoded and the URL paths will be collapsed, removing any extraneous path information. For example, if the remote file name is specified as `ftp://ftp.server.tld/folder1/../folder2/filename.txt` it would be normalized as `ftp://ftp.server.tld/folder2/filename.txt`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddQueuedFile, FtpAddQueuedFileEx, FtpGetFirstQueuedFile, FtpGetNextQueuedFile, FtpGetQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPQUEUESTATUS

This structure is used by the **FtpGetQueueStatus** function to return information about the current status of the file transfer queue.

```
typedef struct _FTPQUEUESTATUS
{
    DWORD dwStatus;
    DWORD dwThreadId;
    DWORD dwQueuedFiles;
    DWORD dwPendingFiles;
    DWORD dwCopiedFiles;
    DWORD dwFailedFiles;
    DWORD dwTimeElapsed;
    DWORD dwCurrentFile;
    DWORD dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    ULARGE_INTEGER uiBytesTotal;
} FTPQUEUESTATUS, *LPFTPQUEUESTATUS;
```

Members

dwStatus

An unsigned integer which specifies the current status of the queue. It can be one of the following values:

Value	Description
FTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before **FtpStartQueue** has been called or after **FtpStopQueue** has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
FTP_QUEUE_STATUS_ACTIVE	Files in the queue are being uploaded or downloaded. The *dwCurrentFile* member of this structure identifies the file which is currently being transferred.
FTP_QUEUE_STATUS_PAUSED	File transfers are currently paused. The queue enters this state after the **FtpSuspendQueue** function is called and resumes file transfers after the **FtpResumeQueue** function is called.

dwThreadId

An unsigned integer which specifies the thread ID for the queue manager which is performing the file transfers. All file transfers are performed asynchronously in a background worker thread. If this structure member is zero, the queue is idle and not performing file transfers. The application should not use the thread ID to obtain a handle to suspend or terminate the thread. This can potentially result in unexpected behavior or instability within the application.

dwQueuedFiles

An unsigned integer which specifies the total number of queued files. This value includes pending and completed file transfers.

dwPendingFiles

An unsigned integer which specifies the number of files which are queued to be transferred.

dwCopiedFiles

An unsigned integer which specifies the number of files which have been successfully transferred. This value reflects the total number of files which have the FTP_QUEUE_FLAG_COMPLETED status.

dwFailedFiles

An unsigned integer which specifies the number of files which have failed to transfer or the file transfer has been canceled. This value reflects the total number of files which have the FTP_QUEUE_FLAG_FAILED status.

dwTimeElapsed

An unsigned integer which specifies the total amount of time, in milliseconds, the queue manager has been active performing a file transfer. If the queue is idle, this value will reflect the total run time for the previously active queue. A value of zero indicates the queue was never active or the queue state has been reset with a call to **FtpResetQueue**.

dwCurrentFile

An unsigned integer which specifies the unique identifier for the current file being transferred. If no file transfer is in progress, this member will have a value of zero. If the value is non-zero, it can be passed to the **FtpGetQueuedFile** function to obtain information about the queued file.

dwLastError

An unsigned integer which specifies the last error code for a failed file transfer. If there have been no errors, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the current or last transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

uiBytesTotal

A value which specifies the total number of bytes which were copied during the current queue run. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftpv12.lib

See Also

FtpGetQueuedFile, FtpGetQueueStatus, FtpCancelQueuedFile, FtpResetQueue, FtpResumeQueue, FtpStartQueue, FtpStopQueue, FtpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUS

This structure is used by the **FtpGetTransferStatus** function to return information about a file transfer in progress.

```
typedef struct _FTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
   TCHAR   szLocalFile[FTP_MAXFILENAMELEN];
   TCHAR   szRemoteFile[FTP_MAXFILENAMELEN];

} FTPTRANSFERSTATUS, *LPFTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

szLocalFile

A null-terminated string which specifies the local file that is being copied to or from the server.

szRemoteFile

A null-terminated string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FTPTRANSFERSTATUSEX** structure which uses 64-bit integers for the file size.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h

See Also

FtpGetTransferStatus, FtpGetTransferStatusEx, FTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUSEX

This structure is used by the **FtpGetTransferStatusEx** function to return information about a file transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPTRANSFERSTATUSEX
{
   ULARGE_INTEGER dwBytesTotal;
   ULARGE_INTEGER dwBytesCopied;
   DWORD          dwBytesPerSecond;
   DWORD          dwTimeElapsed;
   DWORD          dwTimeEstimated;
   DWORD          dwReserved;
   TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
   TCHAR          szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUSEX, *LPFTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

dwReserved

This structure member is reserved for future use.

szLocalFile

A null-terminated string which specifies the local file that is being copied to or from the server.

szRemoteFile

A null-terminated string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option FTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

FtpGetTransferStatus, FtpGetTransferStatusEx, FTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only

	protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library

	will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH

protocol has been specified, this member should be NULL.

lpszKeyFile

A null-terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SFTP (SSH) connections and should always be NULL when establishing a secure connection using FTPS (FTP+TLS). If the member is NULL, then no private key is used. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The ***lpszKeyFile*** path can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results with multithreaded applications because the current working directory for a process is a global value and it can be changed by any thread at any time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol was selected. This protocol will only be used when connecting to an SSH server and is not supported with any other application protocol. Version 1.0 of the protocol is no longer considered secure and is not supported.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description

SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the TLS

protocol.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the TLS protocol.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the TLS protocol.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Server Library

Implements a server that enables the application to send and receive files using the File Transfer Protocol.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSFTSV12.DLL
Version	12.0.3024.1286
LibID	BB27183D-4826-4F44-9EF7-128EADE765D4
Import Library	CSFTSV12.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

This library provides an interface for implementing an embedded, lightweight server that can be used to exchange files with a client using the standard File Transfer Protocol. The server can accept connections from any third-party application or a program developed using the SocketTools FTP client API.

The application specifies an initial server configuration and then responds to events that are raised by the API when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard FTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server supports active and passive mode file transfers, has compatibility options for NAT router and firewall support, and provides support for secure file transfers using explicit TLS. Secure connections require a valid security certificate to be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum

required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Server Functions

Function	Description
FtpAddVirtualUser	Add a new virtual user for the specified server
FtpAuthenticateClient	Authenticate the client and assign access rights for the session
FtpChangeClientDirectory	Change the current working directory for the client session
FtpCreateServerCredentials	Create a new server security credentials structure
FtpDeleteServerCredentials	Delete a previously created security credentials structure
FtpDeleteVirtualUser	Delete a virtual user from the specified server
FtpDisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
FtpEnableClientAccess	Enable or disable access rights for the specified client session
FtpEnableCommand	Enable or disable a specific server command
FtpEnumServerClients	Returns a list of active client sessions established with the specified server
FtpGetActiveClient	Return the client ID for the active client session associated with the current thread
FtpGetClientAccess	Return the access rights that have been granted to the client session
FtpGetClientAddress	Return the IP address of the specified client session
FtpGetClientCredentials	Return the credentials for the specified client session
FtpGetClientDirectory	Return the current working directory for a client session
FtpGetClientFileType	Return the current file type used for transfers by the specified client
FtpGetClientHomeDirectory	Return the home directory for an authenticated client session
FtpGetClientIdentity	Return the identity of the specified client session
FtpGetClientIdleTime	Return the idle timeout period for the specified client
FtpGetClientLocalPath	Return the full local path for the specified virtual path
FtpGetClientServer	Return the handle to the server that created the specified client session
FtpGetClientThreadId	Returns the thread ID associated with the specified client session
FtpGetClientUserName	Return the user name associated with the specified client session
FtpGetClientVirtualPath	Return the virtual path for a local file on the server
FtpGetCommandFile	Return the full path to the local file name or directory specified by the client
FtpGetCommandLine	Return the complete command line issued by the client
FtpGetCommandName	Return the name of the last command issued by the client
FtpGetCommandParam	Return the value of the specified parameter for the command issued by the client
FtpGetCommandParamCount	Return the number of parameters to the current command issued by the client
FtpGetCommandResult	Return the result code and a description of the last command processed by the server
FtpGetCommandUsage	Return the number of times a specific command has been issued by all clients
FtpGetProgramExitCode	Return the exit code of the last program executed by the client
FtpGetProgramName	Return the name of the last program executed by the client
FtpGetProgramOutput	Return a copy of the standard output from the last program executed by the client
FtpGetProgramText	Return a copy of the standard output from the last program in a string buffer
FtpGetRenamedFile	Return the original name of a file being renamed by the client
FtpGetServerAddress	Return the IP address for the server

FtpGetServerDirectory	Return the full path to the root directory assigned to the specified server
FtpGetServerError	Return information about the last server error that occurred
FtpGetServerIdentity	Return the identity and version information for the specified server
FtpGetServerLogFile	Return the current log file format and full path for the file
FtpGetServerMemoryUsage	Return the amount of memory allocated for the server and all client sessions
FtpGetServerName	Return the host name assigned to the server or specified client session
FtpGetServerOptions	Return the configuration options for the specified server
FtpGetServerPriority	Return the current priority assigned to the specified server
FtpGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
FtpGetServerTransferInfo	Return information about the current file transfer for the client session
FtpGetServerUuid	Return the UUID assigned to the specified server
FtpGetServerUuidString	Return the UUID assigned to the server as a printable string
FtpIsClientAnonymous	Determine if the specified client has authenticated as an anonymous user
FtpIsClientAuthenticated	Determine if the specified client session has been authenticated
FtpIsCommandEnabled	Determine if the specified command is currently enabled or disabled
FtpRegisterProgram	Register a program for use with the SITE EXEC command
FtpRenameServerLogFile	Rename or delete the current log file being updated by the server
FtpSendResponse	Send a result code and optional message to the client in response to a command
FtpServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
FtpServerDisableTrace	Disable logging of network function calls
FtpServerEnableTrace	Enable logging of network function calls to a file
FtpServerInitialize	Initialize the library and validates the specified license key at runtime
FtpServerProc	Callback function used to process server events
FtpServerRestart	Restart the server, terminating all active client sessions
FtpServerResume	Resume accepting client connections on the specified server
FtpServerStart	Start the server and begin accepting client connections
FtpServerStop	Stop the server and terminate all active client connections
FtpServerSuspend	Suspend accepting client connections on the specified server
FtpServerThrottle	Limit the number of active client connections, connections per address and connection rate
FtpServerUninitialize	Terminate use of the library by the application
FtpSetClientAccess	Change the access rights associated with the specified client session
FtpSetClientFileType	Change the current file type used for transfers by the specified client
FtpSetClientIdentity	Change the identity string associated with the specified client session
FtpSetClientIdleTime	Change the idle timeout period for the specified client session
FtpSetCommandFile	Change the name of the local file or directory that is the target of the current command
FtpSetServerAddress	Change the IP address that the server will use with passive data connections
FtpSetServerError	Set the last error code for the specified server session
FtpSetServerIdentity	Change the identity and version information for the specified server
FtpSetServerLogFile	Change the current log format, level of detail and file name
FtpSetServerName	Change the hostname assigned to the specified server or client session

FtpSetServerPriority	Change the priority assigned to the specified server
FtpSetServerStackSize	Change the initial size of the stack allocated for threads created by the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAddVirtualUser Function

```
BOOL WINAPI FtpAddVirtualUser(
   HSERVER hServer,
   UINT nHostId,
   DWORD dwUserAccess,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszDirectory
);
```

Add a new virtual user for the specified server.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host. This parameter is reserved for future use and must always have a value of zero.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see User Access Constants.

lpszUserName

A null-terminated string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the function will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A null-terminated string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the function will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

lpszDirectory

A null-terminated string which specifies the directory that will be the virtual user's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created the first time that the virtual user successfully logs in to the server. If this parameter is NULL or an empty string, a default home directory will be created for the virtual user.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **FtpAddVirtualUser** function adds a virtual user that is associated with the specified virtual host. When a client connects with the server and provides authentication credentials, the server will check if the username has been created using this function. If a match is found, the client access rights will be updated.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this function is called with a username that already exists, that record is replaced with the values passed to this function. You cannot use this function to create a virtual user named "anonymous".

The virtual users created by this function exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this function for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpDeleteVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpAuthenticateClient Function

```
BOOL WINAPI FtpAuthenticateClient(
    HSERVER hServer,
    UINT nClientId,
    DWORD dwUserAccess,
    BOOL bCreateHome,
    LPCTSTR lpszDirectory
);
```

Authenticate the client and assign access rights for the session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User Access Constants.

bCreateHome

An integer value that specifics if the server should create the home directory for the authenticated client if it does not already exist. If this value is non-zero, the home directory will be created. If value is zero, the home directory will not be created and if it does not exist, this function will fail.

lpszDirectory

A null-terminated string that specifies the home directory for the user. If an absolute path is specified, it will be relative to the server root directory. If a relative path is specified, it will be a subdirectory of the home directory for the server instance. If this parameter is NULL or an empty string, a home directory will be assigned based on the server home directory and the user name.

Return Value

If the client session could be authenticated, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has already been authenticated, this function will return zero.

Remarks

The **FtpAuthenticateClient** function is used to authenticate a specific client session, typically in response to an FTP_CLIENT_USERAUTH event that indicates a client has requested authentication. This function is also used internally to automatically grant the appropriate access rights to local user and anonymous client sessions.

It is recommended that most applications specify FTP_ACCESS_DEFAULT as the ***dwUserAccess*** value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options for normal and anonymous users. If the server is going to be publicly accessible or third-party FTP clients will be used to access the server, you should always grant the FTP_ACCESS_LIST permission to clients. Many client applications will not function correctly if they are unable to obtain a list of files in the user's home directory.

If FTP_ACCESS_RESTRICTED is specified and the server was started in multi-user mode, the client

session will be effectively locked to its home directory and cannot navigate to the server root directory. By default, restricted client sessions are also limited to only downloading files and requesting directory listings. If a client session is not restricted, the client can access files outside of its home directory. Regardless of this option, a client cannot access files outside of the server root directory.

If FTP_ACCESS_RESTRICTED or FTP_ACCESS_ANONYMOUS is specified, the client session will be authenticated in a restricted mode and the access rights for the session will persist until the client disconnects from the server. Unlike regular users, the access rights for a restricted client cannot be changed by the server at a later point. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

If the *lpszDirectory* parameter is NULL or an empty string and the server has been started in multi-user mode, each user is assigned their own home directory based on their username. If the server has not been started in multi-user mode, then the default home directory will be the server root directory and is shared by all users. The **FtpGetClientHomeDirectory** function will return the full path to the home directory for an authenticated client.

If the FTP_ACCESS_EXECUTE permission is granted to the client session, it can execute external programs using the SITE EXEC command. Because the program is executed in the context of the server process, it is recommended that you limit access to this functionality and ensure that the programs being executed do not introduce any security risks to the operating system. This permission is never granted by default, and the SITE EXEC command will return an error if the client session is anonymous, regardless of whether this permission is granted or not.

This function is should only be used for custom authentication schemes and is not necessary if you have used the **FtpAddVirtualUser** function to create virtual users.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddVirtualUser, FtpChangeClientDirectory, FtpGetClientCredentials, FtpGetClientDirectory, FtpGetClientHomeDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpChangeClientDirectory Function

```
BOOL WINAPI FtpChangeClientDirectory(
    HSERVER hServer,
    UINT nClientId,
    LPCTSTR lpszDirectory
);
```

Change the current working directory for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string which specifies the new current working directory for the client session. If this parameter is NULL or an empty string, the current working directory will be changed to the client home directory. If this parameter is not NULL, it must specify a directory that exists and is accessible by the server process.

Return Value

If the current working directory was changed, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the directory is invalid, this function will return zero.

Remarks

The **FtpChangeClientDirectory** function will change the current working directory for the specified client session. This function is called internally when the client sends the CWD or CDUP commands, however it may be explicitly used by the application to change the client's working directory in response to a server event.

This function cannot be used to change the current working directory for a client to an arbitrary directory outside of the server root directory. If the *lpszDirectory* parameter specifies a relative path (i.e.: a path that does not begin with a drive letter or leading path delimiter) then the new working directory will be relative to the current working directory. If an absolute path is specified, the absolute path must include the complete path to either the server root directory or the user's home directory, based on the permissions granted to the client session. If a path outside of the server root directory is specified, this function will fail with an access denied error.

Use caution when calling this function to override the directory specified by the client when it sends the CWD or CDUP commands. If your application changes the current working directory to one not specified by the client, it may cause unpredictable behavior in the client application because the actual path of the current working directory will not match the directory that was requested.

If this function is used to change the current working directory in response to the CWD command, you should not call the **FtpGetCommandParam** function and pass the command parameter as an argument to this function. You must use the **FtpGetCommandFile** function to obtain the directory name provided by the client prior to calling this function.

The application should never call the **SetCurrentDirectory** function in the Windows API to change the current directory for the process to the working directory of a client session. Because the server is

multithreaded and each client session is managed in its own thread, an application using this library should avoid using relative paths.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientDirectory, FtpGetClientHomeDirectory, FtpGetCommandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpCreateServerCredentials Function

```
BOOL WINAPI FtpCreateServerCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **FtpCreateServerCredentials** function creates a SECURITYCREDENTIALS structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **FtpDeleteServerCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **FtpServerStart** function.

Remarks

The structure that is created by this function may be used as server credentials when accepting a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **FtpServerStart** function.

An application only needs to call this function to support secure connections using TLS. When a certificate is specified, it must have a private key associated with it. Attempting to use a certificate

which does not have a private key will result in an error during the startup process indicating that the server credentials are invalid.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Example

```
// Create the server credentials that identifies the certificate
// that will be used for secure connections
LPSECURITYCREDENTIALS lpSecCred = NULL;

FtpCreateServerCredentials(SECURITY_PROTOCOL_DEFAULT,
                           0,
                           NULL,
                           NULL,
                           lpszCertStore,
                           lpszCertName,
                           NULL,
                           &lpSecCred);

// Start the server
hServer = FtpServerStart(lpszLocalHost,
                         FTP_PORT_DEFAULT,
                         FTP_SERVER_LOCALUSER | FTP_SERVER_UNIXMODE |
FTP_SERVER_SECURE,
                         &ftpConfig,
                         lpEventHandler,
                         0,
                         lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpDeleteServerCredentials, FtpServerStart, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDeleteServerCredentials Function

```
VOID WINAPI FtpDeleteServerCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **FtpDeleteServerCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)
    FtpDeleteServerCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated for the credentials after the server has been started.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpCreateServerCredentials, FtpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDeleteVirtualUser Function

```
BOOL WINAPI FtpDeleteVirtualUser(
   HSERVER hServer,
   UINT nHostId,
   LPCTSTR lpszUserName
);
```

Remove a virtual user from the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host. This parameter is reserved for future use and must always have a value of zero.

lpszUserName

A null-terminated string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This function removes a virtual user that was created by a previous call to the **FtpAddVirtualUser** function. This function will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive. You cannot use this function to delete the "anonymous" user.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAddVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpDisconnectClient Function

```
BOOL WINAPI FtpDisconnectClient(
    HSERVER hServer,
    UINT nClientId
);
```

Close the control connection for the specified client and release the resources allocated for the session

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpDisconnectClient** function will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted. This performs the same operation as if the client sent the QUIT command to the server.

This function sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active file transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this function has returned.

After this function returns, the application should never use the same client ID with another function. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerRestart, FtpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnableClientAccess Function

```
BOOL WINAPI FtpEnableClientAccess(
    HSERVER hServer,
    UINT nClientId,
    DWORD dwUserAccess,
    BOOL bEnable
);
```

Enable or disable access rights for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see User Access Constants.

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **FtpEnableClientAccess** function is used to enable or disable access to specific functionality by the client. The function can only change a single access right and cannot be used to enable or disable multiple access rights in a single function call. To change multiple user access rights for the client, use the **FtpSetClientAccess** function.

This function cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpAuthenticateClient, FtpGetClientAccess, FtpSetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnableCommand Function

```
BOOL WINAPI FtpEnableCommand(
   HSERVER hServer,
   LPCTSTR lpszCommand,
   BOOL bEnable
);
```

Enable or disable a specific server command

Parameters

hServer

The server handle.

lpszCommand

A null-terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this function. This parameter cannot be NULL.

bEnable

An integer value which specifies if the command should be enabled or disabled. If the value is non-zero, the command is enabled. If the value is zero, the command will be disabled.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the command is not recognized, the function will return zero. If the function fails, the **FtpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **FtpEnableCommand** function is used to enable or disable access to a specific command on the server. When a command is disabled, it will also disable any corresponding feature related to that command. For example, if the MDTM command is disabled and a client issues the FEAT command to request a list of supported features, the command will no longer be listed. This function is typically used to enable or disable certain commands for compatibility with older client software. The **FtpIsCommandEnabled** function can be used to determine if a command is enabled or not.

The command name provided to this function must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would bet RETR (retrieve) and STOR (store). Refer to File Transfer Protocol Commands for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the USER and PASS commands are required to perform client authentication and therefore cannot be disabled. If you attempt to disable a required command, this function will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this function affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, either assign the client the appropriate permissions using the **FtpAuthenticateClient** function, or use an event handler to filter the commands.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAuthenticateClient, FtpIsCommandEnabled

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpEnumServerClients Function

```
INT WINAPI FtpEnumServerClients(
    HSERVER hServer,
    DWORD dwReserved
    UINT *lpClients,
    INT nMaxClients
);
```

Return a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

dwReserved

An unsigned integer value that is reserved for future use. This parameter should always be zero.

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the function returns. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the ***lpClients*** array. If the ***lpClients*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetServerError**.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections, the function will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the function with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero.

Example

```
INT nClients = FtpEnumServerClients(hServer, 0, NULL, 0);

if (nClients > 0)
{
    UINT *lpClients = NULL;

    // Allocate memory for an array of client IDs
    lpClients = (UINT *)LocalAlloc(LPTR, nClients * sizeof(UINT));

    if (lpClients == NULL)
    {
        // Virtual memory has been exhausted
        return;
    }
```

```
    nClients = FtpEnumServerClients(hServer, 0, lpClients, nClients);

    if (nClients == FTP_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        TCHAR szUserName[FTP_MAXUSERNAME];

        if (FtpGetClientUserName(hServer, lpClients[nIndex], szUserName,
FTP_MAXUSERNAME))
        {
            // Perform some action with the client user name
        }
    }

    // Free the memory allocated for the client IDs
    LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetActiveClient Function

```
UINT WINAPI FtpGetActiveClient(
    HSERVER hServer
);
```

Return the client ID for the active client session associated with the current thread.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the unique ID associated with the client session for the current thread. If the server handle is invalid or there is no client session active on the current thread, the return value is zero.

Remarks

The **FtpGetActiveClient** function is used to obtain the client ID associated with the current thread. This means this function will only return a client ID if it is called within an event handler or a function called by an event handler. If this function is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpEnumServerClients, FtpGetClientThreadId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientAccess Function

```
BOOL WINAPI FtpGetClientAccess(
   HSERVER hServer,
   UINT nClientId,
   LPDWORD lpdwUserAccess
);
```

Return the access rights that have been granted to the client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwUserAccess

A pointer to an unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see User Access Constants. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **FtpGetClientAccess** function is used to obtain all of the access rights that are currently granted to an authenticated client session. The **FtpEnableClientAccess** function can be used to enable or disable specific permissions, and the **FtpSetClientAccess** function can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;

// Check if the client is a restricted user

if (FtpGetClientAccess(hServer, nClientId, &dwUserAccess))
{
    if (dwUserAccess & FTP_ACCESS_RESTRICTED)
    {
        _tprintf(_T("Client %u authenticated as a restricted user\n"), nClientId);
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpAuthenticateClient, FtpEnableClientAccess, FtpSetClientAccess

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientAddress Function

```
INT WINAPI FtpGetClientAddress(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

Return the IP address of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A null-terminated string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientCredentials Function

```
BOOL WINAPI FtpGetClientCredentials(
   HSERVER hServer,
   UINT nClientId,
   LPFTPCLIENTCREDENTIALS lpCredentials
);
```

Return the user credentials for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **FTPCLIENTCREDENTIALS** structure that will contain information about the user when the function returns. This parameter cannot be NULL.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has not requested authentication, this function will return zero.

Remarks

The **FtpGetClientCredentials** function is used to obtain the username and password that was provided by the client when it requested authentication. Typically this function is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **FtpAuthenticateClient** function to specify that the session has been authenticated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAuthenticateClient, FTPCLIENTCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientDirectory Function

```
INT WINAPI FtpGetClientDirectory(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszDirectory,
   INT nMaxLength
);
```

Returns the current working directory for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string buffer that will contain the current working directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

This function returns the full path to the current working directory for the specified client session. For example, if the server root directory is C:\ProgramData\MyServer and the current working directory for the client is /Research/Documents, this function will return C:\ProgramData\MyServer\Research\Documents as the current working directory for the client session.

It is important to note that the current working directory for client sessions is virtual, and does not reflect the current working directory for the server process. To change the current working directory for a client, use the **FtpChangeClientDirectory** function.

This function should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a current working directory and this function will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

To convert a full path to the virtual path for a specific client session, use the **FtpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpChangeClientDirectory, FtpGetClientHomeDirectory, FtpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientFileType Function

```
BOOL WINAPI FtpGetClientFileType(
    HSERVER hServer,
    UINT nClientId,
    UINT *lpnFileType
);
```

Return the current file type used for transfers by the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpnFileType

A pointer to an unsigned integer value that will contain the current file type used by the client for data transfers. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetClientFileType** function will return the current file type that has been specified by the client sending the TYPE command to the server. The file type determines if there is any conversion performed on the data that is being exchanged between the client and server. The following file types are supported:

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpSetClientFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientHomeDirectory Function

```
INT WINAPI FtpGetClientHomeDirectory(
    HSERVER hServer,
    UINT nClientId,
    LPTSTR lpszDirectory,
    INT nMaxLength
);
```

Returns the home directory for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string buffer that will contain the home directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

This function returns the full path to the home working directory assigned to the specified client session. This will be the same path to the home directory specified when the **FtpAuthenticateClient** function was used to authenticate the client session. If a home directory was not explicitly assigned when the client was authenticated, then this function returns the default home directory that was created for the client, or the server root directory if the FTP_SERVER_MULTIUSER option was not specified when the server was started.

This function should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a home directory and this function will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAuthenticateClient, FtpChangeClientDirectory, FtpGetClientDirectory, FtpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientIdentity Function

```
INT WINAPI FtpGetClientIdentity(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszIdentity,
   INT nMaxLength
);
```

Return the identity of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A null-terminated string buffer that will contain the identity of the client when the function returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero. If the client did not identify itself, this function will return zero.

Remarks

The **FtpGetClientIdentity** function returns the string that the client used to identify itself to the server. The client may use either the CLNT or CSID command to identify itself. Although the CLNT command is considered to be deprecated, it is supported for backwards compatibility with older clients. The identity string does not have any standard format and is used for informational purposes only and does not affect the operation of the server in any way. Not all clients identify themselves, in which case this function will return zero and the ***lpszIdentity*** string buffer will be set to an empty string.

If the client does identify itself, it typically uses the name of the client application that was used to establish the connection. The application may choose to assign an identity to a client session for its own internal purposes using the **FtpSetClientIdentity** function, regardless of whether the client identifies itself.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerIdentity, FtpSetClientIdentity, FtpSetServerIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientIdleTime Function

```
UINT WINAPI FtpGetClientIdleTime(
    HSERVER hServer,
    UINT nClientId,
    UINT *lpnElapsed
);
```

Return the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

A pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL if this information is not required.

Return Value

If the function succeeds, the return value is client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetClientIdleTime** function will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a file data transfer completed. The server will never disconnect a client that is in the process of uploading or downloading a file, regardless of the idle timeout period.

The default idle timeout period for a client is 900 seconds (15 minutes), however the server can be configured to use a different value and individual clients can request the timeout period be changed by sending the IDLE command to the server. For a client to be able to change its own timeout period, it must be granted the FTP_ACCESS_IDLE permission. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). An application can change the timeout period for a specific client session using the **FtpSetClientIdleTime** function.

It is important to note that the idle timeout period only affects authenticated clients. Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server before successfully authenticating with a valid username and password. By default, the authentication timeout period is 60 seconds and is set when the server is started; it cannot be changed for an individual client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpSetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientLocalPath Function

```
INT WINAPI FtpGetClientLocalPath(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   LPTSTR lpszLocalPath,
   INT nMaxLength,
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A null-terminated string that specifies an virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A null-terminated string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **FtpGetClientLocalPath** function takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the current working directory for the client session. This function will recognize a tilde at the beginning of the path to specify the client home directory.

To obtain the virtual path for a local file or directory, use the **FtpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientServer Function

```
HSERVER WINAPI FtpGetClientServer(
    UINT nClientId
);
```

The **FtpGetClientServer** function returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is INVALID_SERVER. To get extended error information, call the **FtpGetServerError** function.

Remarks

The **FtpGetClientServer** function returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this function will fail and return INVALID_SERVER indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

FtpServerAsyncNotify

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientThreadId Function

```
DWORD WINAPI FtpGetClientThreadId(
    HSERVER hServer,
    UINT nClientId
);
```

Returns the thread ID associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is a thread ID. If the function fails, the return value is zero. To get extended error information, call the **FtpGetServerError** function.

Remarks

The **FtpGetClientThreadId** function returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this function to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **FtpDisconnectClient** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpEnumServerClients, FtpGetActiveClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientUserName Function

```
INT WINAPI FtpGetClientUserName(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszUserName,
   INT nMaxLength
);
```

Return the user name associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A null-terminated string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value larger than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetClientUserName** function returns the user name associated with an authenticated client session. If the client has not authenticated itself by sending the USER and PASS commands, this function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAuthenticateClient, FtpGetClientAccess, FtpGetClientHomeDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetClientVirtualPath Function

```
INT WINAPI FtpGetClientVirtualPath(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszLocalPath,
   LPTSTR lpszVirtualPath,
   INT nMaxLength,
);
```

Return the virtual path for a local file on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A null-terminated string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A null-terminated string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is either relative to the server root directory, or the client home directory if the client was authenticated as a restricted user. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the ***lpszLocalPath*** parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this function would return the virtual path to that directory as "/Documents/Research".

If the client session was authenticated as a restricted user, then the virtual path is always relative to the client home directory instead of the server root directory. This is because restricted users are isolated to their own home directory and any subdirectories. For example, if restricted user "John" has a home directory of "C:\ProgramData\MyServer\Users\John" and the ***lpszLocalPath*** parameter was specified as "C:\ProgramData\MyServer\Users\John\Accounting\Projections.pdf" this function would return the virtual path as "/Accounting/Projections.pdf".

If the *lpszLocalPath* parameter specifies a file or directory outside of the server root directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This function can only be used with authenticated clients. If the *nClientId* parameter specifies a client session that has not been authenticated, this function will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandFile, FtpGetClientLocalPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandFile Function

```
INT WINAPI FtpGetCommandFile(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

Get the full path to the local file name or directory specified by the client

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null-terminated and must be large enough to store the complete file path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **FtpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that the command issued by the client accepts a file name as an argument, but the client did not specify one.

Remarks

The **FtpGetCommandFile** function is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard FTP command. For example, if the client sends the RETR command to the server, this function will return the complete path to the local file that the client wants to download. This function will only work with those standard commands that perform some action on a file or directory.

This function should always be used to obtain the file name for a command that performs a file or directory operation. The **FtpGetCommandParam** function will return the actual command parameter, but the file name will typically be relative to the user home directory or server root directory, and cannot be passed directly to a Windows API function. The **FtpGetCommandFile** function normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location.

To change the file or directory name that is the target of the current command, use the **FtpSetCommandFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientDirectory, FtpGetClientHomeDirectory, FtpGetCommandLine, FtpGetCommandParam, FtpSetCommandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandLine Function

```
INT WINAPI FtpGetCommandLine(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszCmdLine,
   INT nMaxLength
);
```

Return the complete command line issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A null-terminated string buffer that will contain the command, including all arguments. The string buffer will be null-terminated and must be large enough to store the complete command line. If this parameter is NULL, the function will return the length of the command line.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **FtpGetServerError** function can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **FtpGetCommandLine** function is used to obtain the command that was issued by the client, and is commonly used inside FTP_CLIENT_COMMAND and FTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively. When the function returns, the string buffer provided by the caller will contain the complete command, including all command parameters. Any extraneous whitespace will be removed, however quoted parameters will be retained as-is.

To obtain a specific parameter to a command, use the **FtpGetCommandParam** function. The **FtpGetCommandParamCount** function will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **FtpGetCommandFile** function should be called to obtain the full path to the specified file rather than using the value of the command parameter.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandFile, FtpGetCommandParam

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandName Function

```
INT WINAPI FtpGetCommandName
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszCommand,
   INT nMaxLength
);
```

Return the name of the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string buffer that will contain the command name. The string buffer will be null-terminated and must be large enough to store the complete command name. If this parameter is NULL, the function will only return the length of the current command in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the ***lpszCommand*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **FtpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **FtpGetCommandName** function is used to obtain the name of the last command that was issued by the client. The command name returned by this function will always be capitalized, regardless of how it was sent by the client. This function is typically used inside FTP_CLIENT_COMMAND and FTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively.

The **FtpGetCommandParam** function can be used to return the value of individual command parameters specified by the client. The **FtpGetComandLine** function can be used to obtain the complete command line issued by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandFile, FtpGetCommandLine, FtpGetCommandParam

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandParam Function

```
INT WINAPI FtpGetCommandParam
   HSERVER hServer,
   UINT nClientId,
   INT nParam,
   LPTSTR lpszParam,
   INT nMaxLength
);
```

Return the value of the specified command parameter from the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nParam

An integer value which specifies the command parameter. A value of zero specifies the command itself, while values greater than zero specify a particular parameter. This function will fail if this value is less than zero or greater than the number of parameters available.

lpszParam

A null-terminated string buffer that will contain the command parameter. The string buffer will be null-terminated and must be large enough to store the complete parameter value. If this parameter is NULL, the function will only return the length of the specified parameter in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the ***lpszParam*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **FtpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **FtpGetCommandParam** function is used to obtain a specific parameter for the last command that was issued by the client. If the parameter was surrounded in quotes, those quotes will be included in the value returned by this function. This function is typically used inside FTP_CLIENT_COMMAND and FTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively.

The **FtpGetCommandParamCount** function will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **FtpGetCommandFile** function should be called to obtain the full path to the specified file rather than using the value of the command parameter. To obtain the complete command line,

use the **FtpGetCommandLine** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandFile, FtpGetCommandLine, FtpGetCommandName, FtpGetCommandParamCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandParamCount Function

```
INT WINAPI FtpGetCommandParamCount
    HSERVER hServer,
    UINT nClientId
);
```

Return the number of command parameters for the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An integer value which specifies the number of parameters that were specified in the last command issued by the client. If the command did not include any parameters, this function will return zero. If the client has not issued a command, or the client session ID is invalid, this function will return -1.

Remarks

The **FtpGetCommandParamCount** function is used to determine the number of parameters specified in the last client command, and the maximum value that may be passed as the parameter index to the **FtpGetCommandParam** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetCommandLine, FtpGetCommandName, FtpGetCommandParam

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandResult Function

```
INT WINAPI FtpGetCommandResult(
    HSERVER hServer,
    UINT nClientId,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

Return the result code and description for the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A null-terminated string buffer that will contain the description of the result code. The string buffer will be null-terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the ***lpszResult*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **FtpGetCommandResult** function is used to determine the result of the last command that was issued by the client and is typically called in the FTP_CLIENT_RESULT event handler. This function should only be called after a command has been processed or the **FtpSendResponse** function has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code, and optionally a text message that further describes the result. The text message may be a single line, or it may span multiple lines, with each line of text terminated by a carriage return and linefeed. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range indicate that the requested action is being initiated, and the client should expect another reply from the server before proceeding. This is normally used with file transfers, indicating to the client that the data transfer has started.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. One exception is the 202 result code which indicates that the command is not implemented, but the client should not consider this to be an error condition.

300-399	Result codes in this range indicate that the requested action cannot complete until additional information is provided to the server. This is normally used with commands that require a specific sequence to complete. For example, the server will send the 331 result code in response to the USER command, which tells the client that it must send the PASS command to complete the authentication process.
400-499	Result codes in this range indicate that the requested action did not take place, but the error condition is temporary and may be attempted again. This error response is usually the result of a failed authentication attempt or a file transfer that could not complete.
500-599	Result codes in this range indicate that the requested action did not take place and the failure is permanent. The client should not attempt to send the command again. This error response is usually the result of an invalid command name, a syntax error or the client not having the appropriate access rights to a resource.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetCommandUsage Function

```
UINT WINAPI FtpGetCommandUsage(
   HSERVER hServer,
   LPCTSTR lpszCommand
);
```

Return the number of times a specific command has been issued by all clients.

Parameters

hServer

The server handle.

lpszCommand

A null-terminated string that specifies a command name. The name is not case-sensitive, but must match a valid server command exactly. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is the number of times the command has been issued by all clients since the server was started. If the server handle or command name is invalid, or the command has never been issued, the return value will be zero.

Remarks

The command name provided to this function must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would bet RETR (retrieve) and STOR (store). Refer to File Transfer Protocol Commands for a complete list of server commands.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEnableCommand, FtpGetCommandLine, FtpGetCommandResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetProgramExitCode Function

```
BOOL WINAPI FtpGetProgramExitCode(
    HSERVER hServer,
    UINT nClientId,
    LPDWORD lpdwExitCode
);
```

Return the exit code of the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwExitCode

A pointer to an unsigned integer that will contain the program exit code when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetProgramExitCode** function returns the exit code of a registered program that was executed by the client using the SITE EXEC command. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by custom programs to communicate status information back to the server application.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetProgramName, FtpGetProgramOutput, FtpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetProgramName Function

```
INT WINAPI FtpGetProgramName(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszProgramName,
   INT nMaxLength
);
```

Return the name of the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A null-terminated string buffer that will contain the name of the last program executed by the client. This parameter cannot be NULL and should be at least 32 characters in size.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the function succeeds, the return value is the length of the command name. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, this function will return zero.

Remarks

The **FtpGetProgramName** function returns the name of the last program that was executed by the client using the SITE EXEC command. The name that is returned is the alias assigned to the program, not the full path to the executable file. The server application would typically use this function in an event handler when processing the FTP_CLIENT_EXECUTE event to determine which program has been executed on behalf of the client. The **FtpGetProgramExitCode** function will return the program's exit code and the **FtpGetProgramOutput** function can be used to obtain a copy of the output generated by the program.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetProgramExitCode, FtpGetProgramOutput, FtpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetProgramOutput Function

```
DWORD WINAPI FtpGetProgramOutput(
    HSERVER hServer,
    UINT nClientId,
    LPBYTE lpBuffer,
    DWORD dwBufferSize
);
```

Return a copy of the standard output from the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the function will return the number of bytes of data that was output by the program. Note that this output is not null-terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the ***lpBuffer*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of bytes copied into the specified buffer. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **FtpGetProgramOutput** function is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of bytes of output available to read, call this function with the ***lpBuffer*** parameter as NULL and the ***dwBufferSize*** parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null-terminated.

This function returns the raw output from the command which may contain escape sequences, control characters and embedded nulls. When the application processes the output returned by this function, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the command as a string, use the **FtpGetProgramText** function.

Example

```
LPBYTE lpBuffer = NULL;  // The output buffer
DWORD cbBuffer = 0;      // Number of bytes in the output buffer

// Determine the number of bytes in the output buffer
cbBuffer = FtpGetProgramOutput(hServer, nClientId, NULL, 0);
```

```
if (cbBuffer > 0)
{
    // Allocate memory for the buffer
    lpBuffer = (LPBYTE)LocalAlloc(LPTR, cbBuffer + 1);

    // Copy the program output to the buffer
    if (lpBuffer != NULL)
        cbBuffer = FtpGetProgramOutput(hServer, nClientId, lpBuffer, cbBuffer + 1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    LocalFree((HLOCAL)lpBuffer);
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetProgramExitCode, FtpGetProgramName, FtpGetProgramText, FtpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetProgramText Function

```
INT WINAPI FtpGetProgramText(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

Return a copy of the standard output from the last program in a string buffer.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the function will return the number of bytes of characters that was output by the program, not including a terminating null character.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the ***lpszBuffer*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **FtpGetProgramText** function is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of characters of output available to read, call this function with the ***lpszBuffer*** parameter as NULL and the ***nMaxLength*** parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This function will only return textual output from the command and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the last command that was executed, use the **FtpGetProgramOutput** function.

Example

```
LPTSTR lpszBuffer = NULL;  // The output buffer
INT cchBuffer = 0;         // Number of bytes in the output buffer

// Determine the number of bytes in the output buffer
cchBuffer = FtpGetProgramText(hServer, nClientId, NULL, 0);

if (cchBuffer > 0)
```

```
{
    // Allocate memory for the string buffer
    lpszBuffer = (LPTSTR)LocalAlloc(LPTR, (cchBuffer + 1) * sizeof(TCHAR));

    // Copy the program output to the buffer
    if (lpszBuffer != NULL)
        cchBuffer = FtpGetProgramText(hServer, nClientId, lpszBuffer, cchBuffer +
1);
}

// Free the memory allocated for the buffer when finished
if (lpszBuffer != NULL)
{
    LocalFree((HLOCAL)lpszBuffer);
    lpszBuffer = NULL;
    cchBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetProgramExitCode, FtpGetProgramName, FtpGetProgramOutput, FtpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetRenamedFile Function

```
INT WINAPI FtpGetRenamedFile(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

Return the original name of a file being renamed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string buffer that will contain the full path of the last file or directory that was renamed. It is recommended that this buffer be at least MAX_PATH characters in size. The value of this parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null characters. This parameter must have a value larger than zero.

Return Value

If the function succeeds, the return value is the length of the file or directory path, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not renamed a file or directory, this function will return zero.

Remarks

When a client wishes to rename a file or directory, it must send two commands in sequence to the server. The first command is RNFR (rename from) which specifies the original name of the file or directory to be renamed. The second command is RNTO (rename to) and must be sent immediately after the RNFR command and specifies the new name for the file or directory. The **FtpGetRenamedFile** function will return the full path of the local file or directory that was specified by the RNFR command. Typically this function is used by an event handler that processes the FTP_CLIENT_COMMAND event to determine the original path name.

This function is only guaranteed to return a meaningful value when called within the context of the FTP_CLIENT_COMMAND event. Calling this function outside of the event handler will return the path of the last renamed file, but there is no way to determine at what point the client issued the command to rename a file or directory. To obtain the new file or directory name, the **FtpGetCommandFile** function should be called from within the event handler.

The RNFR and RNTO commands can also be used to move a file or directory to a new location. For example, they could be used to move a file from one directory to another. An application should never make the assumption that the paths of the original and new file will be the same.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerAddress Function

```
INT WINAPI FtpGetServerAddress(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

Return the IP address of the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A null-terminated string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

This function will return the IP address assigned to the specified server as a printable string. If the ***nClientId*** parameter has a value of zero, this function will return the IP address assigned to the local system. If the FTP_SERVER_NATROUTER option was specified when the server was started, this function will return the external IP address assigned to the system. If the ***nClientId*** parameter specifies a valid client session, this function will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **FtpGetClientAddress** function.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientAddress, FtpGetServerName, FtpSetServerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerDirectory Function

```
INT WINAPI FtpGetServerDirectory(
   HSERVER hServer,
   LPTSTR lpszDirectory,
   INT nMaxLength
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

hServer

The server handle.

lpszDirectory

A null-terminated string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **FtpGetServerDirectory** function will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, a temporary root directory will be created and this function can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This function will return the fully resolved path name, with all environment variables expanded.

There is no corresponding function to change the server root directory after the server has started. To change the root directory, you must stop the server using the **FtpServerStop** function and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerAddress, FtpGetServerIdentity, FtpGetServerName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerError Function

```
DWORD WINAPI FtpGetServerError(
    HSERVER hServer,
    LPTSTR lpszError,
    INT nMaxLength
);
```

Return the last server error code and a description of the error.

Parameters

hServer

The server handle.

lpszError

A null-terminated string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the ***lpszError*** parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **FtpGetServerError** function immediately when a function's return value indicates that an error has occurred. That is because some functions clear the last error code when they succeed.

If the ***hServer*** parameter is specified with a value of INVALID_SERVER, this function will return the last error that occurred for the current thread. This value should only be used when the function does not have access to a valid server handle, such as when the **FtpServerStart** function fails.

It is important to note that the error codes returned by this function are different than the command result codes that are defined in RFC 959, the standard protocol specification for FTP. This function is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **FtpSendResponse** function is used to send result codes to the client, and the **FtpGetCommandResult** function can be used to determine the result of the last command sent by the client.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR. Those functions which clear the last error code when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandResult, FtpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerIdentity Function

```
INT WINAPI FtpGetServerIdentity(
   HSERVER hServer,
   LPTSTR lpszIdentity,
   INT nMaxLength
);
```

Return the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A null-terminated string buffer that will contain the identity of the server when the function returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **FtpGetServerIdentity** function returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number, and is displayed whenever a client establishes its initial connection to the server. The **FtpSetServerIdentity** function can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientIdentity, FtpSetClientIdentity, FtpSetServerIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerLogFile Function

```
BOOL WINAPI FtpGetServerLogFile(
   HSERVER hServer,
   UINT *lpnLogFormat,
   UINT *lpnLogLevel,
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

Return the current log file format and the full path to the file.

Parameters

hServer

The server handle.

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the function returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Value	Description
FTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A null-terminated string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the ***lpszFileName*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **FTPSERVERCONFIG** structure definition for a list of supported log file formats. If the server handle is invalid or logging has not been enabled, this function will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This function can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **FtpRenameServerLogFile** function to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **FtpSetServerLogFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpRenameServerLogFile, FtpSetServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerMemoryUsage Function

```
BOOL WINAPI FtpGetServerMemoryUsage(
   HSERVER hServer,
   ULARGE_INTEGER *lpMemUsage
);
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

hServer

The server handle.

lpMemUsage

A pointer to a ULARGE_INTEGER variable which will specify how much memory has been allocated by the server. This parameter will be initialized to a value of zero by the function and updated with the total number of bytes allocated by the server and all active client sessions when it returns.

Return Value

If the function succeeds, the return value is non-zero and the memory usage value will be updated. If the server handle is invalid, or the server cannot be locked, the return value is zero. Call the **FtpGetServerError** function to determine the cause of the failure.

Remarks

This function returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This function forces the server into a locked state, and all client sessions will block until the function returns. Because this function enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this function can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the function remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetServerStackSize, FtpSetServerStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerName Function

```
INT WINAPI FtpGetServerName(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszHostName,
   INT nMaxLength
);
```

Return the host name assigned to the specified server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will return the host name assigned to the specified server. If the ***nClientId*** parameter has a value of zero, the function will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the ***nClientId*** parameter specifies a client session, then it this function will return the host name that the client used to establish the connection. If the client sends the HOST command to the server, this function will return the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerPriority Function

```
INT WINAPI FtpGetServerPriority(
    HSERVER hServer
);
```

Return the current priority assigned to the specified server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpGetServerPriority** function can be used to determine the current priority assigned to the server. It will return one of the following values:

Value	Description
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

FtpServerStart, FtpSetServerPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerStackSize Function

```
DWORD WINAPI FtpGetServerStackSize(
     SOCKET hServer
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

FtpGetServerMemoryUsage, FtpServerStart, FtpSetServerStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerTransferInfo Function

```
BOOL WINAPI FtpGetServerTransferInfo(
   HSERVER hServer,
   UINT nClientId,
   LPFTPSERVERTRANSFER lpTransferInfo
);
```

Return information about the file transfer that was performed by the server for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **FTPSERVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the ***dwSize*** member of the structure must be initialized to specify the structure size prior to calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function should only be called after the client has issued the APPE, RETR, STOR or STOU commands to initiate a file transfer, otherwise the return value will be zero.

Remarks

The **FtpGetServerTransferInfo** function is used to obtain information about the last file transfer that was performed by the server for the specified client. This function is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandResult, FTPSERVERTRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerUuid Function

```
INT WINAPI FtpGetServerUuid(
   HSERVER hServer,
   UUID *lpUuid
);
```

Return the UUID assigned to the specified server.

Parameters

hServer

The server handle.

lpUuid

A pointer to a UUID structure that will contain the server UUID when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle is invalid or the pointer to the UUID structure is NULL, the function will return a value of zero.

Remarks

The **FtpGetServerUuid** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain a printable string version of the UUID, use the **FtpGetServerUuidString** function.

There is no corresponding function to change the UUID assigned to an active server. The server UUID is assigned when the server is started, and it must be a unique value that is maintained throughout the lifetime of the server. To change the UUID associated with the server, the server must be stopped using the **FtpServerStop** function and another instance of the server started with the new UUID.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetServerUuidString, FTPSERVERCONFIG

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpGetServerUuidString Function

```
INT WINAPI FtpGetServerUuidString(
    HSERVER hServer,
    LPTSTR lpszHostUuid,
    INT nMaxLength
);
```

Return the UUID assigned to the server as a printable string.

Parameters

hServer

The server handle.

lpszHostUuid

A null-terminated string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the buffer is not large enough to store the complete UUID string, the function will return a value of zero.

Remarks

The **FtpGetServerUuidString** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain the numeric UUID value, use the **FtpGetServerUuid** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerUuid, FTPSERVERCONFIG

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsClientAnonymous Function

```
BOOL WINAPI FtpIsClientAnonymous(
    HSERVER hServer,
    UINT nClientId
);
```

Determine if the specified client has authenticated as an anonymous user.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this function will return a non-zero value, otherwise it will return zero. If the server handle and client ID are valid, and the client session has been authenticated, this function will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpAuthenticateClient, FtpGetClientCredentials, FtpIsClientAuthenticated

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsClientAuthenticated Function

```
BOOL WINAPI FtpIsClientAuthenticated(
    HSERVER hServer,
    UINT nClientId
);
```

Determine if the specified client session has been authenticated.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this function will return a non-zero value, otherwise it will return zero. If the server handle and client ID are valid, this function will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpAuthenticateClient, FtpGetClientCredentials, FtpIsClientAnonymous

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpIsCommandEnabled Function

```
BOOL WINAPI FtpIsCommandEnabled(
   HSERVER hServer,
   LPCTSTR lpszCommand
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

hServer

The server handle.

lpszCommand

A null-terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name. Partial matches are not recognized by this function. This parameter cannot be NULL.

Return Value

If the command is enabled, this function will return a non-zero value. If the command is disabled, the server handle is invalid or the command name does not match a supported command, this function will return zero.

Remarks

The **FtpIsCommandEnabled** function is used to determine whether a specific command is enabled. Typically this function is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled and disabled using the **FtpEnableCommand** function.

This function does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **FtpAuthenticateClient** function, and certain commands can be limited by the permissions granted to the client. For example, even though the STOR command is enabled, a client must have the FTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see User Access Constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpAuthenticateClient, FtpEnableCommand, FtpGetCommandName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRegisterProgram Function

```
BOOL WINAPI FtpRegisterProgram(
   HSERVER hServer,
   UINT nHostId,
   LPCTSTR lpszCommandName,
   LPCTSTR lpszProgramFile,
   LPCTSTR lpszParameters,
   LPCTSTR lpszDirectory
);
```

Register a program for use with the SITE EXEC command.

Parameters

hServer

The server handle.

nHostId

An unsigned integer that identifies the virtual host associated with the program. This value should always be zero.

lpszCommandName

A null-terminated string which specifies the name of the site specific command. This is the name that is passed to the SITE EXEC command and does not need to match the actual name of the executable file on the local system. The maximum length of the command name is 32 characters. This parameter cannot be NULL.

lpszProgramFile

A null-terminated string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A null-terminated string that specifies additional parameters for the program. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A null-terminated string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the current working directory of the client that issues the SITE EXEC command.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpRegisterProgram** function registers an executable program for use with the SITE EXEC command. Because this can present a significant security risk to the server, clients are not given permission to use this command by default. A client must be explicitly granted permission to use SITE EXEC by including FTP_ACCESS_EXECUTE as one of the permissions when authenticating the client session with the **FtpAuthenticateClient** function or creating a virtual user using the **FtpAddVirtualUser** function.

To give the server complete control over what programs can be executed using SITE EXEC, the

program must be registered with the server and referenced by an alias specified by the *lpszCommandName* parameter. The maximum length of a program name is 31 characters and it must be at least 3 characters in length. The name must only consist of alphanumeric characters and the first character of the program name cannot be numeric. The program name is not case-sensitive, however convention is to use upper-case characters. If a program name is specified that already has been registered, it will be updated with the new information provided by this function.

The *lpszProgramFile* string specifies file name of the program that will be executed. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple command names that reference the same executable file. The only requirement is that the command names be unique. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string is used to define optional command line parameters that will be included with the command. This string can contain placeholders that are replaced by additional parameters specified by the client when it sends the SITE EXEC command. First replacement parameter is %1, the second is %2 and so on.

The executable program that is registered using this function must be a console application that writes to standard output. Programs that write directly to a console, or programs written to use a Windows user interface are not supported and will yield unpredictable results. In most cases, those programs that do not use standard input and output will be forcibly terminated by the server. If the program attempts to read from standard input, it will immediately encounter an end-of-file condition. Programs executed by the SITE EXEC command have no input; it is similar to a program that has its input redirected from the NUL: device. If the program must process a file on the server, the local file name should be passed as a command line parameter.

The output from the program will be redirected back to the client control channel. The output should be textual, with each line of text terminated by a carriage return and linefeed (CRLF). Programs that write binary data to standard output, particular data with embedded nulls, will yield unpredictable results and are not supported. To ensure that the program output conforms to the protocol standard, any non-printable characters will be replaced with a space and each line of output will be prefixed by a single space. The server application can obtain a copy of the output from the last command by calling the **FtpGetProgramOutput** function.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetProgramExitCode, FtpGetProgramName, FtpGetProgramOutput

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpRenameServerLogFile Function

```
BOOL WINAPI FtpRenameServerLogFile(
    HSERVER hServer,
    LPCTSTR lpszFileName
);
```

Rename or delete the current log file being updated by the server.

Parameters

hServer

The server handle.

lpszFileName

A null-terminated string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero. If logging is not currently enabled for the server, this function will return zero.

Remarks

The **FtpRenameServerLogFile** function is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the ***lpszFileName*** parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This function must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **FtpGetServerLogFile** function to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** functions with that file.

To disable logging, use the **FtpSetServerLogFile** function and specify the logging format as FTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerLogFile, FtpSetServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSendResponse Function

```
BOOL WINAPI FtpSendResponse(
   HSERVER hServer,
   UINT nClientId,
   UINT nResultCode,
   LPCTSTR lpszMessage
);
```

Send a result code and message to the client in response to a command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nResultCode

An unsigned integer value which specifies the result code.

lpszMessage

A null-terminated string which specifies a message to be sent to the client. If this parameter is NULL or points to an empty string, a default message associated with the result code will be used.

Return Value

If the result code and message text was sent to the client, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the result code is invalid, this function will return zero.

Remarks

The **FtpSendResponse** function is used to respond to a command issued by the client. Command responses are normally handled by the server as a normal part of processing a command and this function is only used if the application has implemented custom commands or wishes to modify the standard responses sent by the server. The message may be a maximum of 2048 characters and may include embedded carriage-return and linefeed characters. If no message is specified, then a default message will be sent based on the result code.

Result codes must be three digits (in the range of 100 through 999) and although this function will support the use of non-standard result codes, it is recommended that the client application use the standard codes defined in RFC 959 whenever possible. The use of non-standard result codes may cause problems with FTP clients that expect specific result codes in response to a particular command. For more information, refer to the **FtpGetCommandResult** function.

This function should only be called once in response to a command sent by the client. If a result code has already been sent in response to a command and this function is called, it will fail and return a value of zero. This is necessary because sending multiple result codes in response to a single command may cause unpredictable behavior by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerAsyncNotify Function

```
BOOL WINAPI FtpServerAsyncNotify(
    HSERVER hServer,
    HWND hWnd,
    UINT uMsg
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hServer

The server handle.

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **FtpGetServerError** function.

Remarks

The **FtpServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the ***hWnd*** parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the ***hWnd*** parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The ***wParam*** message parameter will contain the notification message and the ***lParam*** message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Value	Description
FTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the **FtpServerStart** function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The ***lParam*** message parameter will specify the handle to the server.
FTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the

	application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the **FtpServerResume** function to resume accepting client connections. The *lParam* message parameter will specify the handle to the server.
FTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the **FtpServerSuspend** function. This notification message may be sent to the application multiple times over the lifetime of the server. The *lParam* message parameter will specify the handle to the server.
FTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the **FtpServerRestart** function. Note that the server socket handle provided by the *lParam* message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The *lParam* message parameter will specify the handle to the server.
FTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The *lParam* message parameter will specify the unique ID of the client that connected to the server.
FTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the **FtpServerStop** function. The *lParam* message parameter will specify the unique ID of the client that disconnected from the server.
FTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in *lParam* will reference the defunct server and cannot be used with other server functions. The *lParam* message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The FTP_NOTIFY_CONNECT and FTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **FtpGetClientServer** function to obtain a handle to the server that created the client

session. Note that at the time the application processes the FTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

FtpGetClientServer, FtpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerDisableTrace Function

```
BOOL WINAPI FtpServerDisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerEnableTrace Function

```
BOOL WINAPI FtpServerEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the process level and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpServerDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerInitialize Function

```
BOOL WINAPI FtpServerInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **FtpServerInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **FtpServerInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **FtpServerUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **FtpServerInitialize** or **FtpServerUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpServerStart, FtpServerStop, FtpServerUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerProc Function

```
VOID CALLBACK FtpServerProc(
   HSERVER hServer,
   UINT nClientId,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **FtpServerProc** function is an application-defined callback function that processes events generated by the server process.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see Server Event Constants.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero. The **FtpGetServerError** function can be used to obtain additional information about the error code.

dwParam

A user-defined integer value which was specified when the server was started. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

Return Value

None.

Remarks

The FTP_CLIENT_COMMAND event can be used to filter commands issued by the client. If the command performs an operation on a file or directory, the **FtpGetCommandFile** function can be used to obtain the file name on the server that the client specified. It is possible for the application to change the file name using the **FtpSetCommandFile** function to direct the server to use a different file than the one specified by the client. If the file name is changed and the event handler returns a value of zero, the server will perform the default action for the command using the new file name.

The FTP_CLIENT_TIMEOUT event will only occur for authenticated client sessions. The server has a second, internal timer that is maintained for unauthenticated sessions that requires the client to successfully authenticate itself within a fixed period of time. If a client does not authenticate within the allowed time period, the server will automatically close the control channel. For more information, refer to the documentation for the *nAuthTime* member of the **FTPSERVERCONFIG** structure.

The callback function will be called in the context of the thread that is currently managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the

results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within an event handler. Controls should only be modified by the same UI thread that created their window. To change the user interface in response to a server event, use the **FtpServerAsyncNotify** function to enable asynchronous notifications and update the UI in response to the notification message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerAsyncNotify, FtpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerRestart Function

```
HSERVER WINAPI FtpServerRestart(
    HSERVER hServer
);
```

Restart the server, terminating all active client sessions.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the new handle for the specified server. If the function fails, the return value is INVALID_SERVER. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerRestart** function will restart the specified server, terminating all active client sessions. The server handle that is returned by the function is the handle for the new server instance, and the old handle value is no longer valid. If the function is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **FtpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerStart, FtpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerResume Function

```
BOOL WINAPI FtpServerResume(
    HSERVER hServer
);
```

Resume accepting client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerResume** function instructs the server to resume accepting new client connections after the **FtpServerSuspend** function has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerRestart, FtpServerStart, FtpServerStop, FtpServerSuspend, FtpServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerStart Function

```
HSERVER WINAPI FtpServerStart(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   DWORD dwOptions,
   LPFTPSERVERCONFIG lpServerConfig,
   FTPSERVPROC lpServerProc,
   DWORD_PTR dwServerParam,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **FtpServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit TLS. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies the options used when creating an instance of the server. For a list of options, see Server Option Constants.

lpServerConfig

A pointer to an **FTPSERVERCONFIG** structure that specifies the configuration options for the server. If this parameter is NULL, then default configuration parameters will be used that starts the server in a restrictive mode.

lpServerProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **FtpServerProc** callback function. If this parameter is NULL, a default internal handler is used to process client commands.

dwServerParam

A user-defined integer value that is passed to the callback function. If the *lpServerProc* parameter is NULL, this value should be zero. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

lpCredentials

Pointer to a **SECURITYCREDENTIALS** structure. If this parameter is NULL, the default security credentials for the server host name will be used. If security is enabled for the server, it is

recommended that you provide a pointer to this structure with specific information about the server certificate that should be used. If the security options are not specified in the server configuration, this parameter is ignored.

Return Value

If the function succeeds, the return value is a handle to a server session. If the function fails, the return value is INVALID_SERVER. To get extended error information, call **FtpGetServerError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **FtpServerStop** function.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

If the server is started with security enabled, the *lpCredentials* parameter should be used to provide the server certificate information in a **SECURITYCREDENTIALS** structure. If the pointer is NULL, this function will search for a certificate that matches the hostname provided as part of the server configuration. This requires that the server certificate be installed in the personal certificate store of the current process user, and it must have a private key associated with it. To use a certificate with a different name, or one that is stored in a PFX file, a **SECURITYCREDENTIALS** structure must be defined and passed to this function.

Example

```
HSERVER hServer;
FTPSERVERCONFIG ftpConfig;

// Initialize the server configuration
ZeroMemory(&ftpConfig, sizeof(FTPSERVERCONFIG));
ftpConfig.dwSize = sizeof(ftpConfig);
ftpConfig.nMaxClients = 20;
ftpConfig.nMaxGuests = 5;
ftpConfig.nLogFormat = FTP_LOGFILE_EXTENDED;
ftpConfig.nAuthTime = 30;
```

```
ftpConfig.nIdleTime = 120;
ftpConfig.lpszIdentity = _T("MyProgram");
ftpConfig.lpszHostName = _T("server.company.com");
ftpConfig.lpszHostUuid = _T("10000000-1000-1000-1000-100000000000");
ftpConfig.lpszDirectory = _T("%ProgramData%\\MyProgram\\Files");
ftpConfig.lpszLogFile = _T("%ProgramData%\\MyProgram\\Server.log");

// Start the server
hServer = FtpServerStart(lpszLocalHost,
                         FTP_PORT_DEFAULT,
                         FTP_SERVER_LOCALUSER | FTP_SERVER_UNIXMODE,
                         &ftpConfig,
                         lpEventHandler,
                         0,
                         NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpEnumServerClients, FtpServerProc, FtpServerRestart, FtpServerStop, FTPSERVERCONFIG, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerStop Function

```
BOOL WINAPI FtpServerStop(
   HSERVER hServer
);
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **FtpServerStop** function instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **FtpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerRestart, FtpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerSuspend Function

```
BOOL WINAPI FtpServerSuspend(
    HSERVER hServer
);
```

Suspend the server and reject new client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **FtpServerResume** function. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **FtpServerThrottle** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerRestart, FtpServerResume, FtpServerStart, FtpServerStop, FtpServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerThrottle Function

```
BOOL WINAPI FtpServerThrottle(
   HSERVER hServer,
   UINT nMaxClients,
   UINT nMaxClientsPerAddress,
   UINT nMaxGuests,
   DWORD dwConnectionRate
);
```

The **FtpServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server.

nMaxClients

An integer value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

An integer value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxGuests

An integer value which specifies the maximum number of guest users that may login to the server. A value of zero disables guest logins and requires that all clients authenticate with a valid username and password. A value of -1 specifies that the maximum number of guest users should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client

sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **FtpServerStart** function is called with the maximum number of clients set to 100, and then **FtpServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerRestart, FtpServerResume, FtpServerStart, FtpServerSuspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpServerUninitialize Function

```
VOID WINAPI FtpServerUninitialize();
```

The **FtpServerUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **FtpServerInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **FtpServerUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **FtpServerUninitialize** for every successful call to **FtpServerInitialize** made by a process. Operations for all threads in the server are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpServerInitialize, FtpServerStart, FtpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetClientAccess Function

```
BOOL WINAPI FtpSetClientAccess(
    HSERVER hServer,
    UINT nClientId,
    DWORD dwUserAccess
);
```

Change the access rights associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User Access Constants.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **FtpSetClientAccess** function can change the access rights for an authenticated client session. This function can only be used after the **FtpAuthenticateClient** function has been used to grant the initial set of access rights to the client. The **FtpEnableClientAccess** function can be used to grant or revoke a specific permission for the client session.

The *dwUserAccess* parameter has a value of FTP_ACCESS_DEFAULT, then default permissions will be granted to the client session. A normal client cannot be changed to a restricted or anonymous client using this function. If the FTP_ACCESS_RESTRICTED or FTP_ACCESS_ANONYMOUS access flags are specified, this function will fail.

This function cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Example

```
DWORD dwUserAccess = 0;

// Allow the client to execute programs using SITE EXEC
if (FtpGetClientAccess(hServer, nClientId, &dwUserAccess))
{
    if (! (dwUserAccess & FTP_ACCESS_ANONYMOUS))
        FtpSetClientAccess(hServer, nClientId, dwUserAccess | FTP_ACCESS_EXECUTE);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpAuthenticateClient, FtpEnableClientAccess, FtpGetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetClientFileType Function

```
BOOL WINAPI FtpSetClientFileType(
    HSERVER hServer,
    UINT nClientId,
    UINT nFileType
);
```

Change the current file type used for transfers by the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nFileType

Specifies the type of file that will be uploaded or downloaded. This parameter determines whether subsequent file transfers require any data conversion and may be one of the following values.

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpSetClientFileType** function will change the default file type that is used for subsequent transfers by the client. If the file type is set to FILE_TYPE_ASCII then the server will automatically convert any end-of-line character sequences to match the format used by the local system. For example, if the client is connecting from a UNIX based system, the server will convert a single linefeed character to a carriage return (CR) and linefeed (LF) sequence. If the file type is set to FILE_TYPE_IMAGE, then no conversion is performed.

This function can be used to override the file type specified by filtering the TYPE command issued by the client. For example, it could be used to force file transfers to use a specific type based on the file extension, regardless of the type specified by the client. To determine the current file type set by the client, use the **FtpGetClientFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetClientFileType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetClientIdentity Function

```
BOOL WINAPI FtpSetClientIdentity(
    HSERVER hServer,
    UINT nClientId,
    LPCTSTR lpszIdentity
);
```

Change the identity of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A null-terminated string that identifies the client. If this parameter is NULL or specifies an empty string, the current identity for the client is cleared. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the server handle or client ID is invalid, the function will return a value of zero.

Remarks

The **FtpSetClientIdentity** function associates a string value with the client that can be used to identify the session. The identity string does not have any standard format and is used for informational purposes only. Typically it is used to identify the client application that was used to establish the connection. Changing the client identity has no effect on the operation of the server. To obtain the identity string currently associated with the client, use the **FtpGetClientIdentity** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientIdentity, FtpGetServerIdentity, FtpSetServerIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetClientIdleTime Function

```
UINT WINAPI FtpSetClientIdleTime(
    HSERVER hServer,
    UINT nClientId,
    UINT nTimeout
);
```

Change the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the function succeeds, the return value is the previous client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpSetClientIdleTime** function is will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **FtpGetClientIdleTime** function.

This timeout period only affects authenticated clients. Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server and that value cannot be changed for individual client sessions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetCommandFile Function

```
BOOL WINAPI FtpSetCommandFile(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszFileName
);
```

Change the name of the local file or directory that is the target of the current command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpSetCommandFile** function is used by the application to change the target file or directory name for the current command from within an FTP_EVENT_COMMAND event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the RETR command to download a file from the server, this function can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **FtpGetCommandFile** function.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is . If the path is relative, it will be relative to the current working directory for the client session. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this function to change the target file name.

Typically this function would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this function to provide access to data that is stored outside of the server root directory. Incorrect use of this function could expose the server to security risks or cause unpredictable behavior by client applications.

This function should only be called within the context of the FTP_EVENT_COMMAND event, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_COMMAND. To obtain the name of the current command issued by the client, use the **FtpGetCommandName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetCommandFile FtpGetCommandLine, FtpGetCommandName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerAddress Function

```
BOOL WINAPI FtpSetServerAddress(
   HSERVER hServer,
   LPCTSTR lpszAddress
);
```

Change the IP address that the server will use with passive data connections.

Parameters

hServer

The server handle.

lpszAddress

A null-terminated string that specifies the IP address that the server should use for passive mode data transfers. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle or the IP address is invalid the function will return a value of zero.

Remarks

The **FtpSetServerAddress** function changes the IP address that the server will use when a client transfers a file using a passive mode data connection. In passive mode, the server will create a second passive (listening) socket that will accept an incoming connection from the client. The server sends the IP address and port number allocated for that socket to the client, and the client establishes the data connection to the server using that address. The IP address that the server sends to the client is normally the same as the IP address that the client used to establish the control connection, however if the server is located behind a router that performs Network Address Translation (NAT), the IP address reported to the client may not be usable.

This function enables your application to set the external IP address for the server to a specific value, rather than the server attempting to automatically discover its own external address. If you wish to set the external address for the server manually, call the **FtpServerStart** function without the FTP_SERVER_EXTERNAL option and then call this function to set the external IP address to the desired value.

This function will not change the IP address the server is using to listen for client connections. The only way to change the listening IP address is to stop and restart the server using the new address. This function only changes the IP address that is reported to clients when a passive data connection is used. Incorrect use of this function can prevent the client from establishing a data connection to the server. The address must be in the same address family as the local address that the server was started with. For example, if the server was started using an IPv4 address, the IP address passed to this function cannot be an IPv6 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerAddress, FtpGetServerName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerError Function

```
VOID WINAPI FtpSetServerError(
    HSERVER hServer,
    DWORD dwError
);
```

Set the last error code for the specified server session.

Parameters

hServer

The server handle.

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR.

If the ***hServer*** parameter is specified as INVALID_SERVER, this function will set the last error code for the current thread, but will not change the error code associated with any server session. This should only be done if the application does not have access to a valid server handle.

If the ***dwError*** parameter is specified with a value of zero, this effectively clears the error code for the last function that failed. Those functions which clear the last error code when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **FtpGetServerError** function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetCommandResult, FtpGetServerError, FtpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerIdentity Function

```
BOOL WINAPI FtpSetServerIdentity(
   HSERVER hServer,
   LPCTSTR lpszIdentity
);
```

Change the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A null-terminated string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **FtpSetClientIdentity** function changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **FtpGetServerIdentity** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientIdentity, FtpSetClientIdentity, FtpGetServerIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerLogFile Function

```
BOOL WINAPI FtpSetServerLogFile(
   HSERVER hServer,
   UINT nLogFormat,
   UINT nLogLevel,
   LPCTSTR lpszFileName
);
```

Change the current log format, level of detail and file name.

Parameters

hServer

The server handle.

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Value	Description
FTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of FTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A null-terminated string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero.

Remarks

The **FtpSetServerLogFile** function can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the ***nLogFormat*** parameter with the constant FTP_LOGFILE_DELETE.

The higher the value of the ***nLogLevel*** parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record file transfers, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = FTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (FtpGetServerLogFile(hServer, &nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != FTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = FtpSetServerLogFile(hServer, nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerLogFile, FtpRenameServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerName Function

```
BOOL WINAPI FtpSetServerName(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszHostName
);
```

Change the host name assigned to the specified server or client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will change the host name assigned to the specified client session. If the ***nClientId*** parameter has a value of zero, the function will change default host name that was assigned to the server as part of the server configuration. If the ***nClientId*** parameter specifies a valid client session and the ***lpszHostName*** parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **FtpGetServerName** function will return the host name specified by the client, and **FtpSetServerName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerAddress, FtpGetServerName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerPriority Function

```
INT WINAPI FtpSetServerPriority(
    HSERVER hServer,
    INT nPriority
);
```

Change the priority assigned to the specified server.

Parameters

hServer

The server handle.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Value	Description
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Import Library: csftsv12.lib

See Also

FtpGetServerPriority, FtpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FtpSetServerStackSize Function

```
BOOL WINAPI FtpSetServerStackSize(
    SOCKET hServer,
    DWORD dwStackSize
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csftsv12.lib

See Also

FtpGetServerMemoryUsage, FtpGetServerStackSize, FtpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

File Transfer Protocol Server Data Structures

- FTPCLIENTCREDENTIALS
- FTPSERVERCONFIG
- FTPSERVERTRANSFER
- SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPCLIENTCREDENTIALS

The **FTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _FTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    TCHAR szHostName[FTP_MAXHOSTNAME];
    TCHAR szUserName[FTP_MAXUSERNAME];
    TCHAR szPassword[FTP_MAXPASSWORD];
} FTPCLIENTCREDENTIALS, *LPFTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value reserved for future use. This member will always be initialized to a value of zero.

szHostName

A null-terminated string that specifies the server host name.

szUserName

A null-terminated string that specifies the user name.

szPassword

A null-terminated string that specifies the user password.

Remarks

When an instance of this structure is passed to the **FtpGetClientCredentials** function, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetClientCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPSERVERCONFIG

The **FTPSERVERCONFIG** structure provides the configuration information used to create an instance of an FTP server.

```
typedef struct _FTPSERVERCONFIG
{
    DWORD   dwSize;
    UINT    nMaxClients;
    UINT    nMaxClientsPerAddress;
    UINT    nMaxGuests;
    UINT    nLogFormat;
    UINT    nLogLevel;
    UINT    nMinPort;
    UINT    nMaxPort;
    UINT    nAuthFail;
    UINT    nAuthTime;
    UINT    nIdleTime;
    UINT    nExecTime;
    LPCTSTR lpszIdentity;
    LPCTSTR lpszHostUuid;
    LPCTSTR lpszHostName;
    LPCTSTR lpszRootPath;
    LPCTSTR lpszTempPath;
    LPCTSTR lpszLogFile;
} FTPSERVERCONFIG, *LPFTPSERVERCONFIG;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

nMaxClients

An integer value that specifies the maximum number of active client connections that will be accepted by the server. If the maximum number of clients is reached, any further connections are rejected by the server until one or more clients close their connection to the server or are disconnected. A value of zero specifies the default configuration value of 100 connections should be used.

nMaxClientsPerAddress

An integer value that specifies the maximum number of active client connections per IP address that will be accepted by the server. If the maximum number of clients from the same IP address is reached, any further connections are rejected until one of the clients closes its connection to the server. A value of zero specifies the default configuration value of 4 connections per address should be used.

nMaxGuests

An integer value that specifies the maximum number of anonymous client connections. If the server does not permit anonymous connections, this value is ignored. It is recommended that this value always be less than the value of the ***nMaxClients*** member. A value of zero specifies that the default configuration limit of 20 anonymous connections should be used.

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Value	Description
FTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If the ***nLogFormat*** member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format. The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

nMinPort

An integer value that specifies the minimum range of port numbers that will be used with passive data connections. A value of zero specifies that the default value of 30000 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be less than the value of the ***nMaxPort*** structure member.

nMaxPort

An integer value that specifies the maximum range of port numbers that will be used with passive data connections. A value of zero specifies the default value of 65535 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be greater than the value of the ***nMinPort*** structure member.

nAuthFail

An integer value that specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies that the default configuration limit of 3 authentication attempts per login should be allowed. The maximum number of authentication attempts is 10.

nAuthTime

An integer value that specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 20 seconds and the maximum value is 300 seconds (5 minutes). This value is used to ensure that a client has successfully authenticated itself within a limited period of time. This prevents a potential denial-of-service attack against the server where clients establish connections and hold them open without authentication. In conjunction with the ***nAuthFail*** member, this also limits the ability of a client to attempt to probe the server for valid username and password combinations.

nIdleTime

An integer value that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default

value of 900 seconds (15 minutes). If the value is non-zero, the minimum value is 60 seconds and the maximum value is 7200 seconds (2 hours). This value is used to initialize the default idle timeout period for each client session. A client may request that the server change the idle timeout period for its session by sending the SITE IDLE command. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress.

nExecTime

An unsigned integer value that specifies the maximum number of seconds that an external program is permitted to run on the server. External programs are registered using the **FtpRegisterProgram** function, and are executed by the client sending the SITE EXEC command to the server. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds.

lpszIdentity

A null-terminated string that identifies the server. It is used for informational purposes only and has no effect on the operation of the server. If this member is not initialized to a value, then a default identity will be used. The server identity is provided when a client establishes the initial connection and when the client sends the STAT or CSID commands. If this member is defined, it is recommended that you only use printable ASCII characters.

lpszHostUuid

A null-terminated string that specifies a Universally Unique Identifier (UUID) that is used to uniquely identify the server. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This structure member may be initialized to an empty string, in which case a temporary UUID will be randomly generated.

lpszHostName

A null-terminated string that specifies the fully-qualified host name for the server. If this structure member is initialized with an empty string, the local host name assigned to the Windows system will be used. This value does not need to correspond to the actual host name associated with the server's IP address. This value is used for informational purposes only and has no effect on the operation of the server.

lpszRootPath

A null-terminated string that specifies the path to the root directory for the server. If this structure member is initialized with an empty string, the current working directory for the process will be used as the root directory. It is recommended that you provide a full path to an existing directory and do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszTempPath

A null-terminated string that specifies the path to the temporary directory for the server. If this structure member is initialized with an empty string, the current temporary directory for the process will be used. The temporary directory cannot be the same as the root directory, and it is strongly recommended that you do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszLogFile

A null-terminated string that specifies the name of the server log file to create, if a logging format has been specified. If logging is enabled and this member is an empty string, then a default log file name will be created. If the file name does not include a path, then the file is created in the server log directory. If the file name includes a path, the log file will be created using that specific name. If the server is in multi-user mode, then the default location for log files will be the Logs subdirectory

in the server root directory. If the server is not in multi-user mode, the default location for log files will be the temporary directory for the server process.

Remarks

When an instance of this structure is passed to the **FtpServerStart** function, the *dwSize* member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the configuration is invalid.

The *nMaxClients* member limits the total number of client connections and the *nMaxGuests* member limits the total number of guest connections. The server will notify the client if either number of connections has been exceeded and will automatically close the connection. However, one difference is that a new client that exceeds the maximum number of connections will not generate any events; a client that exceeds the maximum number of guest connections will generate events. The reason for the difference is that the check for the number of active clients is made immediately after the client connects to the server, prior to session context being created for that client. The check for the number of anonymous clients is made later, after the client has sent the USER and PASS commands to the server. For more control over how the server accepts client connections, use the **FtpServerThrottle** function.

If the server is running on a system that is behind a firewall that restricts the range of port numbers for inbound connections, the *nMinPort* and *nMaxPort* members can be used to limit the ports which the server will use to listen for passive data connections from clients. By default, the server will use data port numbers that range from 30000 to 65535, which is typical for many firewall configurations. If you provide your own port range, the minimum number of ports available must be at least 1000. Specifying a port range that is too small will cause the **FtpServerStart** function to fail with an error indicating that the server configuration is invalid.

When specifying the root directory for the server configuration, it is important to consider that a client's access to files on the system will be limited to these directories and any subdirectories. If a root directory is not specified, then one will be created using the current working directory for the process. The root directory path may include environment variables surrounded by % symbols and these will be expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

For servers that are publicly accessible, or where you want files to be a accessible across multiple server sessions, you should always populate the *szHostUuid* member with a valid UUID string, and the *szDirectory* member should specify an absolute path to an existing directory.

If the FTP_SERVER_MULTIUSER option is specified, the server will start in multi-user mode, with each user assigned to their own home directory based on their username. This option will cause the server to create a directory structure in the root directory that includes three subdirectories: Public, Users and Logs. The Public directory is the home directory for anonymous users, and they are isolated to that directory. The Users directory is where the user home directories are created. The Logs directory is where log files are created by default. If this option is not specified, then no subdirectories are created and all users will share the server root directory by default.

If the FTP_SERVER_LOCALUSER option is specified, the server will never attempt to authenticate the local Administrator or Guest accounts. This is done to prevent clients from attempting to probe for the administrator account password. However, the application can override this behavior by implementing

a handler for the FTP_CLIENT_LOGIN event.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpServerStart, FtpServerStop, FtpServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FTPSERVERTRANSFER

The **FTPSERVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _FTPSERVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} FTPSERVERTRANSFER, *LPFTPSERVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Value	Description
FTP_FILE_READ	The file was opened for reading. This mode indicates that the client issued the RETR command to download the contents of a file from the server to the client system. The ***szFileName*** member specifies the name of the local file on the server that was downloaded by the client.
FTP_FILE_WRITE	The file was opened for writing. This mode indicates that the client issued the STOR or STOU command to upload the contents of a file from the client system to server. The ***szFileName*** member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.
FTP_FILE_APPEND	The file was opened for writing. This mode indicates the client issued the APPE command to upload the contents of a file from the client system and append the data to a file on the server. If the file did not exist, then it was created. The ***szFileName*** member specifies the name of the local file that was appended to or created by the client.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds.

This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a ULARGE_INTEGER union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file. If the client specified a restart offset using the REST command, this value would only represent the number of bytes transferred from that byte offset, not the total file size.

szFileName

A null-terminated string value that will contain the full path to the local file that was transferred. The ***dwFileAccess*** member determines whether the file name represents a file that was downloaded by the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **FtpGetServerTransferInfo** function, the ***dwSize*** member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

FtpGetServerTransferInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify the security protocol and server certificate when accepting a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Client Library

Transfer files between the local system and a web server, execute scripts and perform remote file management functions.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSHTPV12.DLL
Version	12.0.3024.1286
LibID	E5D90316-2B5F-4E36-BD24-F58F1B47C3BF
Import Library	CSHTPV12.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 7230, RFC 7540

Overview

The Hypertext Transfer Protocol (HTTP) is a lightweight, stateless application protocol that is used to access resources on web servers, as well as send data to those servers for processing. The library provides direct, low-level access to the server and the commands that are used to retrieve resources (i.e.: documents, images, etc.). The library also provides a simple interface for downloading resources to the local host, similar to how the HTTP library can be used to download files.

In a typical session, you would use the class to establish a connection, send a request (to download a resource, post data for processing, etc.), read the data returned by the server and then disconnect. It is the responsibility of the client to process the data returned by the server, depending on the type of resource that was requested.

This library supports HTTP versions 1.0, 1.1, 2.0 and 3.0. Secure connections use TLS 1.2 or later and supports the use of client certificates if required. There are several functions designed to make it easy to submit JSON and XML requests and include custom headers which may be required by REST APIs. The library also supports standard authentication methods and the use of OAuth bearer tokens.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum

required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Functions

Function	Description
HttpAddFormField	Add the form field and its value to the specified form.
HttpAddFormFile	Add the contents of the file to the specified form.
HttpAddQueuedFile	Add a new file to the queue to be uploaded or downloaded.
HttpAddQueuedFileEx	Add a new file to the queue with additional transfer options.
HttpAddRequestHeaders	Add one or more request headers for the current client session.
HttpAsyncConnect	Establish an asynchronous connection to the specified server.
HttpAsyncGetData	Copy the contents of a resource on the server to a local buffer.
HttpAsyncGetFile	Copy a file from the server to the local system.
HttpAsyncGetFileEx	Copy a file from the server to the local system, use with files larger than 4 GiB.
HttpAsyncPostData	Post data from a local buffer to a script executed on the server.
HttpAsyncPostJson	Post JSON formatted data to a script executed on the server.
HttpAsyncPostXml	Post XML formatted data to a script executed on the server.
HttpAsyncProxyConnect	Establish an asynchronous connection to a proxy server.
HttpAsyncPutData	Create a file on the server using the contents of a local buffer.
HttpAsyncPutFile	Copy a file from the local system to the server.
HttpAsyncPutFileEx	Copy a file from the local system to the server, use with files larger than 4 GiB.
HttpAsyncSubmitForm	Submit the specified form to the server for processing.
HttpAttachThread	Attach the specified client handle to another thread.
HttpAuthenticate	Specify authentication information for restricted resources.
HttpCancel	Cancel the current blocking operation.
HttpCancelQueuedFile	Cancel the queued file transfer.
HttpClearForm	Remove all defined fields from the specified form.
HttpClearHeaders	Resets the current request and response headers to their default values.
HttpClearQueue	Remove all files from the current transfer queue.
HttpCloseFile	Close the file opened on the server.
HttpCommand	Send a command to the server.
HttpConnect	Connect to the specified server.
HttpConnectUrl	Establish a client connection using the specified URL.
HttpCreateFile	Create or replace a file on the server.
HttpCreateForm	Create a new form and return the form handle.
HttpCreateQueue	Create a new file transfer queue.
HttpCreateSecurityCredentials	Create a new security credentials structure.

HttpDeleteFile	Remove a file from the server.
HttpDeleteFormField	Delete the form field and its value from the specified form.
HttpDeleteHeaders	Delete all of the response or request headers for the current session.
HttpDeleteQueue	Delete a previously created file transfer queue.
HttpDeleteSecurityCredentials	Delete a previously created security credentials structure.
HttpDestroyForm	Destroy the specified form and free the memory allocated for it.
HttpDisableEvents	Disable asynchronous event notification.
HttpDisableTrace	Disable logging of network function calls.
HttpDisconnect	Disconnect from the current server.
HttpDownloadFile	Download a file from the server to the local system.
HttpEnableCompression	Enable or disable support for data compression.
HttpEnableEvents	Enable asynchronous event notification.
HttpEnableTrace	Enable logging of network function calls to a file.
HttpEnumQueuedFiles	Return a list of files in the transfer queue.
HttpEnumTasks	Return a list of asynchronous tasks.
HttpEventProc	Callback function that processes events generated by the client.
HttpFindQueuedFile	Find a file in the transfer queue which matches the search criteria.
HttpFreezeEvents	Suspend asynchronous event processing.
HttpGetBearerToken	Return the current OAuth 2.0 bearer token for the client session.
HttpGetContentLength	Return the size of the current resource in bytes.
HttpGetContentType	Return the content type for the current resource.
HttpGetCookie	Return information about the specified cookie.
HttpGetData	Copy the specified resource to a local buffer.
HttpGetDefaultUserAgent	Return the default user agent string included with requests.
HttpGetDefaultZoneId	Return the default security zone for downloaded files.
HttpGetEncodingType	Determines which content encoding option is enabled.
HttpGetErrorString	Return a description for the specified error code.
HttpGetFile	Copy a file from the server to the local system.
HttpGetFileEx	Copy a file from the server to the local system, use with large files over 4 GiB.
HttpGetFilePathUrl	Return a normalized URL for the specified file on the server.
HttpGetFileSize	Return the size of a file on the server.
HttpGetFileSizeEx	Return the size of a file on the server, supports large files over 4 GiB.
HttpGetFileTime	Return the date and time a file on the server was last modified.
HttpGetFileZoneInfo	Return file zone identifier information for the specified local file.
HttpGetFirstCookie	Return the first cookie set by the server.

HttpGetFirstHeader	Return the name and value of the first request or response header field.
HttpGetFirstQueuedFile	Return information about the first file in the transfer queue.
HttpGetFormProperties	Return the properties of the specified form.
HttpGetLastError	Return the last error code.
HttpGetNextCookie	Return the next cookie set by the server.
HttpGetNextHeader	Return the name and value of the next request or response header field.
HttpGetNextQueuedFile	Return information about the next file in the transfer queue.
HttpGetOption	Return the enabled/disabled state of a specified option.
HttpGetPriority	Return the current priority for file transfers.
HttpGetQueueClient	Return the handle to the current client session.
HttpGetQueuedFile	Return information about a file queued for transfer.
HttpGetQueueStatus	Return the status of the current file transfer queue.
HttpGetQueueThread	Return a value which uniquely identifies the queue manager thread.
HttpGetRedirectUrl	Return the URL for a redirected resource.
HttpGetResourceUrl	Return the URL for the current request.
HttpGetRequestHeader	Return the value of the specified request header field.
HttpGetResponseHeader	Return the value of the specified response header field.
HttpGetResultCode	Return the result code from the previous command.
HttpGetResultString	Return the result string from the previous command.
HttpGetSecurityInformation	Return security information about the current client connection.
HttpGetSecurityZoneId	Returns the security zone identifier for the specified resource.
HttpGetStatus	Return the current client status.
HttpGetTaskError	Return the last error code for the specified asynchronous task.
HttpGetTaskId	Return the unique task identifier associated with the specified client session.
HttpGetText	Return the contents of a text resource to a string buffer.
HttpGetTextEx	Return the contents of a text resource with additional conversion options.
HttpGetThreadQueue	Return the handle to a queue associated with the specified thread.
HttpGetTimeout	Return the number of seconds until an operation times out.
HttpGetTransferStatus	Return data transfer statistics.
HttpInitialize	Initialize the library and validate the specified runtime license key.
HttpIsBlocking	Determine if the client is blocked, waiting for information.
HttpIsConnected	Determine if the client is connected to the server.
HttpIsReadable	Determine if data can be read from the server.
HttpIsWritable	Determine if data can be written to the server.
HttpNormalizePath	Normalize the specified resource returning a canonical path.

HttpNormalizeUrl	Normalize the specified URL returning a canonical path.
HttpOpenFile	Open a file on the server for reading.
HttpPatchData	Submits JSON or XML patch data to the server and returns the response .
HttpPostData	Submit data to the server using the POST command and returns the response.
HttpPostFile	Submit the contents of a local file to the server.
HttpPostJson	Submit JSON structured data to the server and returns the response.
HttpPostText	Submit plain text data to the server and returns the response.
HttpPostTextEx	Submit plain text data to the server with additional conversion options.
HttpPostXml	Submit XML structured data to the server and returns the response.
HttpProxyConnect	Establish a connection with the specified proxy server.
HttpPutData	Create a file on the server using the contents of a local buffer.
HttpPutDataEx	Submit data to the server using the PUT command and returns the response.
HttpPutFile	Copy a file from the local system to the server.
HttpPutFileEx	Copy a file from the local system to the server, use with large files over 4 GiB.
HttpPutText	Create a text file on the server from the contents of a string buffer.
HttpPutTextEx	Submit text to the server using the PUT command and returns the response.
HttpRead	Read data from the server.
HttpRegisterEvent	Register an event callback function.
HttpRemoveQueuedFile	Remove a file from the transfer queue.
HttpResetQueue	Reset the internal state of file transfers in the specified queue.
HttpResumeQueue	Resume the transfer of files after queue processing has been paused.
HttpSetBearerToken	Set the value of the OAuth 2.0 bearer token for the client session.
HttpSetContentType	Set the content type for the next request.
HttpSetCookie	Set the value of the specified cookie.
HttpSetDefaultUserAgent	Set the default user agent string included with requests.
HttpSetDefaultZoneId	Set the default security zone for downloaded files.
HttpSetEncodingType	Specifies the type of encoding to be applied to data submitted to the server.
HttpSetFileZoneInfo	Set or remove the zone indentifier associated with the specified file.
HttpSetFormProperties	Modify the properties of the specified form.
HttpSetLastError	Set the last error code.
HttpSetOption	Enable or disable the specified option.
HttpSetPriority	Set the priority for file transfers.
HttpSetRequestHeader	Set the value of a request header field.
HttpSetTimeout	Set the number of seconds until an operation times out.
HttpStartQueue	Begin transferring the files in the specified queue.

HttpStopQueue	Stop transferring the files in the specified queue.
HttpSubmitForm	Submit the specified form to the server for processing.
HttpSubmitRequest	Submit a request to the server and return the response.
HttpSuspendQueue	Pause the transfer of files in the specified queue.
HttpTaskAbort	Abort the specified asynchronous task.
HttpTaskDone	Determine if an asynchronous task has completed.
HttpTaskResume	Resume execution of an asynchronous task.
HttpTaskSuspend	Suspend execution of an asynchronous task.
HttpTaskWait	Wait for an asynchronous task to complete.
HttpUninitialize	Terminate use of the library by the application.
HttpUploadFile	Upload a file from the local system to the server.
HttpValidateHostName	Validate the specified host name and return the resolved IP address.
HttpValidateUrl	Check the contents of a string to ensure it represents a valid URL.
HttpVerifyFile	Compare the size of a local file against a file stored on the server.
HttpWaitForQueue	Wait for the transfer of all queued files to complete.
HttpWrite	Write data to the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddFormField Function

```
INT WINAPI HttpAddFormField(
   HFORM hForm,
   LPCTSTR lpszFieldName,
   LPVOID lpFieldData,
   DWORD cbFieldData,
   DWORD dwReserved
);
```

The **HttpAddFormField** function adds a new field to the specified form.

Parameters

hForm

Handle to the virtual form.

lpszFieldName

A null-terminated string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpFieldData

The form field data. Typically this will either by a pointer to an array of bytes or a string which specifies the value for the form field. If no data is to be associated with the form field, then this argument may be NULL.

cbFieldData

An unsigned integer value which specifies the number of bytes of data in the form field. If this value is -1 then it is assumed that the ***lpFieldData*** parameter is a pointer to a null-terminated string. If ***lpFieldData*** is NULL, this value must be zero.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpAddFormField** function is used to add a field and its associated value to a form created using the **HttpCreateForm** function. If the field name has already been added to the form, the previous value is deleted and replaced by the new value.

If the Unicode version of this function is called and the ***cbFieldData*** parameter is -1, the ***lpFieldData*** parameter must point to a wide string terminated with a null character. The string will be automatically converted to UTF-8 when the form data is submitted to the server.

Example

```
HFORM hForm = INVALID_FORM;
HGLOBAL hgblResult = (HGLOBAL)NULL;
DWORD cbResult = 0;
INT nResult = 0;

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);
```

```
if (hForm == INVALID_FORM)
    return;

HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nResult = HttpSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);
HttpDestroyForm(hForm);

if (hgblResult != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer points to data returned by the server after the form
    // data was submitted

    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddFormFile, HttpClearForm, HttpCreateForm, HttpDeleteFormField, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddFormFile Function

```
INT WINAPI HttpAddFormFile(
   HFORM hForm,
   LPCTSTR lpszFieldName,
   LPCTSTR lpszFileName,
   DWORD dwReserved
);
```

The **HttpAddFormFile** function adds the contents of a file to the specified form.

Parameters

hForm

Handle to the virtual form.

lpszFieldName

A null-terminated string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpszFileName

A null-terminated string which specifies the name of the file. The contents of the file will be added to the form data that is submitted to the server.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddFormField, HttpClearForm, HttpCreateForm, HttpDeleteFormField, HttpPostFile, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddQueuedFile Function

```
DWORD WINAPI HttpAddQueuedFile(
   HQUEUE hQueue,
   DWORD dwQueueMode,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   UINT nTimeout
);
```

The **HttpAddQueuedFile** function adds a new file to the queue with additional transfer options.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An integer value which specifies if the file should be downloaded from the server or uploaded to the server. This parameter must be one of the following options and cannot have a value of zero.

Constant	Description
HTTP_QUEUE_DOWNLOAD	The file should be queued for downloading from the server to the local system.
HTTP_QUEUE_UPLOAD	The file should be queued for uploading from the local system to the server.

lpszLocalFile

A null-terminated string which specifies the name of the local file to be queued for transfer. This parameter cannot be NULL or point to an empty string. If the file is being queued for an upload, the file must exist on the local system or the function will fail. If the file is being downloaded, it will be created or replaced on the local system.

lpszRemoteFile

A null-terminated string which specifies the URL to the location of the file on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator and can be any of the options used with the **HttpConnect** function. If this value is zero, the default options for the queue will be used. The most common options are:

Value	Description
HTTP_OPTION_DEFAULT	Default options should be used for the queued transfer. All standard headers will be included with the request and redirected resources will return an error, allowing the application to determine if it should request the resource from the new location or inform the user the resource location has changed.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle

	resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.

nTimeout

The number of seconds to wait for a response from the server before failing the file transfer. If this value is zero, the default timeout period for the queue will be used.

Return Value

If the function succeeds, the return value is a unique identifier which corresponds to the queued file. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function will add a file to the current transfer queue. If successful, the value returned is a unique integer which identifies that file in the queue. To obtain information about the queued file, you can call the **HttpGetQueuedFile** function and provide this return value as the ***dwFileId*** parameter. To remove a file from the queue, use the **HttpRemoveQueuedFile** function.

Files can only be added to the queue when in an idle or paused state. If you attempt to add a file to the queue while the queue manager is transferring files, the function will fail. To determine the current state of the queue, call the **HttpGetQueueStatus** function

There are no fixed limits to the number of files which can be queued for transfer. To release the memory allocated for the queue, call the **HttpClearQueue** function. To determine the current status of the queue, including how many file transfers have been queued and how many are pending completion, call the **HttpGetQueueStatus** function.

If the local file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

This function will not immediately begin the process of uploading or downloading the file. It will only add the transfer request to the current queue. The file transfers are managed by a background worker thread which is created when the **HttpStartQueue** function is called.

The **HttpAddQueuedFileEx** function is an extended version of this function with additional parameters to specify a hostname, port number, user name and password. Because most file uploads require authentication, using the extended function may be preferred to composing a URL which includes the account credentials.

Example

```
// Create a new queue
HQUEUE hQueue = HttpCreateQueue(INFINITE, HTTP_TIMEOUT, HTTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = HttpAddQueuedFile(
                     hQueue,
                     lpszLocalFile,
                     lpszRemoteFile,
                     HTTP_OPTION_DEFAULT,
                     HTTP_TIMEOUT);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (HttpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (HttpStartQueue(hQueue, HTTP_QUEUE_ALL, 0, NULL, 0))
{
    HttpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
    HttpStopQueue(hQueue);
}

// Remove all files from the queue
HttpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFileEx, HttpCreateQueue, HttpDeleteQueue, HttpFindQueuedFile, HttpGetFirstQueuedFile, HttpGetNextQueuedFile, HttpGetQueuedFile, HttpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddQueuedFileEx Function

```
DWORD WINAPI HttpAddQueuedFileEx(
    HQUEUE hQueue,
    DWORD dwQueueMode,
    LPCTSTR lpszHostName,
    UINT nHostPort,
    UINT nAuthType,
    LPCTSTR lpszUserName,
    LPCTSTR lpszPassword,
    LPCTSTR lpszLocalFile,
    LPCTSTR lpszRemoteFile,
    DWORD dwOptions,
    UINT nTimeout,
    LPVOID lpvReserved
);
```

The **HttpAddQueuedFileEx** function adds a new file to the queue with additional transfer options.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An integer value which specifies if the file should be downloaded from the server or uploaded to the server. This parameter must be one of the following options and cannot have a value of zero.

Constant	Description
HTTP_QUEUE_DOWNLOAD	The file should be queued for downloading from the server to the local system.
HTTP_QUEUE_UPLOAD	The file should be queued for uploading from the local system to the server.

lpszHostName

The name of the server to establish a connection with. This may be a fully-qualified domain name or an IP address. If this parameter is NULL or points to an empty string, the ***lpszRemoteFile*** parameter must specify a complete URL which identifies the server as well as the path to the remote file.

nHostPort

The port number which should be used when establishing the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443. If this parameter is zero and a URL is specified with the ***lpszRemoteFile*** parameter, the port number included in the URL will be used as the default value.

nAuthType

An unsigned integer value which specifies the method to be used when authenticating the client. The following values are supported:

Value	Description
HTTP_AUTH_NONE	No client authentication should be performed. The ***lpszUserName***

	and ***lpszPassword*** parameters are ignored and current authentication settings are cleared.
HTTP_AUTH_BASIC	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the connection. If this parameter is NULL or an empty string, no client credentials are provided. If the authentication type is HTTP_AUTH_BEARER, this parameter is ignored.

lpszPassword

A null-terminated string which specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified. If the authentication type is HTTP_AUTH_BEARER, this value must be the bearer token issued by the service provider.

lpszLocalFile

A null-terminated string which specifies the name of the local file to be queued for transfer. This parameter cannot be NULL or point to an empty string. If the file is being queued for an upload, the file must exist on the local system or the function will fail. If the file is being downloaded, it will be created or replaced on the local system.

lpszRemoteFile

A null-terminated string which specifies the name of the file on the server. If the file is being downloaded, the file must exist on the server or the transfer will fail. If the file is being uploaded, the file will be created or replaced on the server. This parameter may specify a complete URL. Note that any values you specify as arguments to this function will override the values specified in the URL. If you want to use a URL as the remote file name, the ***lpszHostName*** parameter should be NULL and the ***nHostPort*** parameter should be zero.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator and can be any of the options used with the **HttpConnect** function. If this value is zero, the default options for the queue will be used. The most common options are:

Value	Description
HTTP_OPTION_DEFAULT	Default options should be used for the queued transfer. All standard headers will be included with the request and redirected resources will return an error, allowing the application to determine if it should request the resource from the new location or inform the user the resource location has changed.

HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.

nTimeout

The number of seconds to wait for a response from the server before failing the file transfer. If this value is zero, the default timeout period for the queue will be used.

lpvReserved

A reserved parameter which should always be NULL.

Return Value

If the function succeeds, the return value is a unique identifier which corresponds to the queued file. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function will add a file to the current transfer queue. If successful, the value returned is a unique integer which identifies that file in the queue. To obtain information about the queued file, you can call the **HttpGetQueuedFile** function and provide this return value as the ***dwFileId*** parameter. To remove a file from the queue, use the **HttpRemoveQueuedFile** function.

Files can only be added to the queue when in an idle or paused state. If you attempt to add a file to the queue while the queue manager is transferring files, the function will fail. To determine the current state of the queue, call the **HttpGetQueueStatus** function

There are no fixed limits to the number of files which can be queued for transfer. To release the memory allocated for the queue, call the **HttpClearQueue** function. To determine the current status of the queue, including how many file transfers have been queued and how many are pending completion, call the **HttpGetQueueStatus** function.

If the local file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

This function will not immediately begin the process of uploading or downloading the file. It will only add the transfer request to the current queue. The file transfers are managed by a background worker thread which is created when the **HttpStartQueue** function is called.

Example

```
// Create a new queue
HQUEUE hQueue = HttpCreateQueue(INFINITE, HTTP_TIMEOUT, HTTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = HttpAddQueuedFileEx(
                    hQueue,
                    HTTP_QUEUE_DOWNLOAD,
                    lpszHostName,
                    HTTP_PORT_DEFAULT,
                    HTTP_AUTH_DEFAULT,
                    lpszUserName,
                    lpszPassword,
                    lpszLocalFile,
                    lpszRemoteFile,
                    HTTP_OPTION_DEFAULT,
                    HTTP_TIMEOUT,
                    NULL);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (HttpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}

// Start the queued file transfers and wait for it to complete
if (HttpStartQueue(hQueue, HTTP_QUEUE_ALL, 0, NULL, 0))
{
    HttpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
    HttpStopQueue(hQueue);
}
```

```
// Remove all files from the queue
HttpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpCreateQueue, HttpDeleteQueue, HttpFindQueuedFile, HttpGetFirstQueuedFile, HttpGetNextQueuedFile, HttpGetQueuedFile, HttpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddRequestHeaders Function

```
BOOL WINAPI HttpAddRequestHeaders(
   HCLIENT hClient,
   LPCTSTR lpszHeaderList
);
```

The **HttpAddRequestHeaders** function adds one or more request headers for the current client session.

Parameters

hClient

Handle to the client session.

lpszHeaderList

A null-terminated string which specifies one or more header values which should be set for the current client session. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpAddRequestHeaders** function enables your application to set one or more header values by providing a list of name/value pairs separated by a colon. Multiple header values may be provided by separating them with a newline character. This function is similar to calling the **HttpSetRequestHeader** function for each value. When the list of header values is parsed, extraneous whitespace is ignored; however, if the header list contains invalid text (for example, a missing colon separating a header name from its value) the function will fail and an error will be returned.

Exercise caution when providing a header list created directly from user input, such as a list of values input using a textbox control. Any header values which have been previously set by your application can be overridden by this function and may yield unpredictable results. If the service you are using requires a custom authorization header, such as an API token or other user credentials, allowing users to directly modify request header values this way can present a security risk.

Example

```
// Define a list of header values which should be included with the
// request submitted to the server
LPCTSTR lpszHeaderList = _T("X-API-Token: 99d2fe39-0246-4efa-98e0-4d775579fa5d\n")
\
                         _T("X-API-UserId: someuser@domain.tld\n") \
                         _T("X-API-Version: 1.1\n")

if (!HttpAddRequestHeaders(hClient, lpszHeaderList))
{
    // Unable to add the headers for this client session
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetFirstHeader, HttpGetNextHeader, HttpGetRequestHeader, HttpGetResponseHeader, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncConnect Function

```
HCLIENT WINAPI HttpAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg

);
```

The **HttpAsyncConnect** function is used to establish a connection with the specified server

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **HttpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically

	handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.

HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the client will default to using TLS 1.2 or later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create a version value and the following values are defined:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.
HTTP_VERSION_09	The client should use the original one-line protocol which

	includes no version number and no request header block. This version has been deprecated and should only be used with legacy servers which do not support the protocol standard. This version of the protocol does not support virtual hosts and is not supported by most modern web services.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.
HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. If the client or server does not support HTTP/2, the client will automatically attempt to use an earlier version of the protocol.
HTTP_VERSION_30	The client should use the HTTP/3 protocol standard defined in RFC 9114. This protocol version is similar to earlier versions of the protocol in terms the request methods and server responses, but the underlying transport mechanism is significantly different. Instead of using TCP, this version of the protocol uses multiplexed UDP connections designed to reduce the overhead of TCP while maintaining reliability. If the client or server does not support HTTP/3, the client will automatically attempt to use an earlier version of the protocol.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event occurs when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

Most applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Under some circumstances, a server may forcibly close or abort a connection if the request fails. For example, this can happen when a malformed request is submitted to the server or a server-side error occurs. When this happens, the application will need to disconnect from the server and establish a new connection, even if the keep-alive option has been specified.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return INVALID_CLIENT.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

The handle returned by this function is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **HttpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncProxyConnect, HttpConnect, HttpDisconnect, HttpInitialize, HttpProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncGetData Function

```
UINT WINAPI HttpAsyncGetData(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Copies the contents of a resource on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource on the server that will be transferred to the local system.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is copied to the local system exactly as it is stored on the server.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **HttpEnableCompression**

	function.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncGetData** function is used to download the contents of a server resource into a local buffer. This function is similar to the **HttpGetData** function, however it retrieves the contents of the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being copied to the local system.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, such as with the example listed below, then you must ensure that your code does not return from the function while the data is being downloaded. In the example, this is achieved by calling the **HttpTaskWait** function. You can also perform other operation and poll the status of the task by calling the **HttpTaskDone** function. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the ***lpvBuffer*** and ***lpdwLength*** parameters on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

The ***lpvBuffer*** parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the file or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the ***lpdwLength*** parameter must be initialized to the maximum number of bytes that

can be copied into the buffer. If specifies the address of a global memory handle, then *lpdwLength* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;
UINT nTaskId;

// Copy the data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter

nTaskId = HttpAsyncGetData(hClient,
                           lpszResource,
                           &hgblBuffer,
                           &dwLength,
                           HTTP_TRANSFER_DEFAULT,
                           NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetFile, HttpAsyncPutData, HttpAsyncPutFile, HttpEventProc, HttpGetFile, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncGetFile Function

```
UINT WINAPI HttpAsyncGetFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Downloads the specified file from the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is copied to the local system exactly as it is stored on the server.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **HttpEnableCompression** function. This option is ignored if the ***dwOffset*** parameter is non-zero.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically

used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncGetFile** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **HttpGetFile** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Example

```
UINT nTaskId;

// Begin a file transfer in the background
```

```
nTaskId = HttpAsyncGetFile(hClient,
                           lpszLocalFile,
                           lpszRemoteFile,
                           HTTP_TRANSFER_DEFAULT,
                           0, NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetData, HttpAsyncPutData, HttpAsyncPutFile, HttpEventProc, HttpGetFile, HttpTaskDone, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncGetFileEx Function

```
UINT WINAPI HttpAsyncGetFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is copied to the local system exactly as it is stored on the server.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **HttpEnableCompression** function. This option is ignored if the ***dwOffset*** parameter is non-zero.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies

that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncGetFileEx** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **HttpGetFileEx** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncPutFileEx, HttpEventProc, HttpGetFileEx, HttpPutFileEx, HttpTaskDone, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncPostData Function

```
UINT WINAPI HttpAsyncPostData(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPVOID lpvBuffer,
   DWORD cbBuffer,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Post data from a local buffer to a script executed on the server.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource that the data will be posted to on the server. Typically this is the name of a script that will be executed. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

The data that will be provided to the script. This parameter may be NULL if the script does not require any additional data from the client.

cbBuffer

The number of bytes to copy from the buffer. If this ***lpvBuffer*** parameter is NULL, this value should be zero.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is

	automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_MULTIPART	The contents of the buffer being sent to the server consists of multipart form data and will be sent as-is without any encoding.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPostData** function is used to submit data to a script that executes on the server and then copy the output from that script into a local buffer. This function is similar to the **HttpPostData** function, however it submits the data using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent and the response from the server is being returned to the caller.

Because this function works asynchronously, it is important that the memory allocated for the *lpvBuffer* and *lpvResult* parameters will not be released before the asynchronous task completes. If you provide pointers to memory that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

If you wish to submit XML formatted data to the server, it is recommended that you use the **HttpAsyncPostXml** function instead.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;
UINT nTaskId;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nTaskId = HttpAsyncPostData(hClient,
                            lpszResource,
                            lpParameters,
                            cbParameters,
                            &hgblBuffer,
                            &cbBuffer,
                            HTTP_POST_CONVERT,
                            NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetData, HttpAsyncPostJson, HttpAsyncPostXml, HttpEventProc, HttpPostData, HttpTaskWait

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncPostJson Function

```
UINT WINAPI HttpAsyncPostJson(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszJsonData,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Submit JSON formatted data to the server and return the response to the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name that the JSON data will be submitted to. Typically this is the name of a script on the server.

lpszJsonData

A null-terminated string that specifies the JSON data that will be submitted to the server.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPostJson** function is used to submit JSON formatted data to a script that executes on the server and then copy the output from that script into a local buffer. It automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure that it is well-formed. Your application is responsible for ensuring that the JSON data that is being submitted to the server is formatted correctly. This function is similar to the **HttpPostJson** function, however it submits the data using a background worker thread and does not block the current working thread.

Because this function works asynchronously, it is important that the memory allocated for the *lpszJsonData* and *lpvResult* parameters will not be released before the asynchronous task completes. If you provide pointers to memory that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;
UINT nTaskId;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nTaskId = HttpAsyncPostJson(hClient,
                            lpszResource,
                            lpszJsonData,
                            &hgblBuffer,
                            &cbBuffer,
                            HTTP_POST_CONVERT,
                            NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetData, HttpAsyncPostData, HttpEventProc, HttpPostJson, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncPostXml Function

```
UINT WINAPI HttpAsyncPostXml(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszXmlData,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Submit XML formatted data to the server and return the response to the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name that the XML data will be submitted to. Typically this is the name of a script on the server.

lpszXmlData

A null-terminated string that specifies the XML data that will be submitted to the server.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPostXml** function is used to submit XML formatted data to a script that executes on the server and then copy the output from that script into a local buffer. It automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure that it is well-formed. Your application is responsible for ensuring that the XML data that is being submitted to the server is formatted correctly. This function is similar to the **HttpPostXml** function, however it submits the data using a background worker thread and does not block the current working thread.

Because this function works asynchronously, it is important that the memory allocated for the *lpszXmlData* and *lpvResult* parameters will not be released before the asynchronous task completes. If you provide pointers to memory that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;
UINT nTaskId;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nTaskId = HttpAsyncPostXml(hClient,
                           lpszResource,
                           lpszXmlData,
                           &hgblBuffer,
                           &cbBuffer,
                           HTTP_POST_CONVERT,
                           NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetData, HttpAsyncPostData, HttpEventProc, HttpPostXml, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncProxyConnect Function

```
HCLIENT WINAPI HttpAsyncProxyConnect(
   UINT nProxyType,
   LPCTSTR lpszProxyHost,
   UINT nProxyPort,
   LPCTSTR lpszProxyUser,
   LPCTSTR lpszProxyPassword,
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **HttpAsyncProxyConnect** function is used to establish a connection through a proxy server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **HttpProxyConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

nProxyType

An unsigned integer which specifies the type of proxy that the client is connecting to. The supported proxy server types are as follows:

Value	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the *dwOptions* parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server settings configured in Windows.

lpszProxyHost

A null-terminated string which specifies the proxy server host name or IP address. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

nProxyPort

The port number that the proxy server is listening for connections on. A value of zero specifies that the default port number 80 should be used. Note that in most cases, a proxy server is not configured to use the default port. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

lpszProxyUser

A null-terminated string which specifies the user name that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszProxyPassword

A null-terminated string which specifies the password that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszRemoteHost

A null-terminated string which specifies the name of the server to connect to through the proxy server. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection

	and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the proxy server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS

	protocol and the client will default to using TLS 1.2 or later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create a version value and the following values are defined:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.
HTTP_VERSION_09	The client should use the original one-line protocol which includes no version number and no request header block. This version has been deprecated and should only be used with legacy servers which do not support the protocol standard.

	This version of the protocol does not support virtual hosts and is not supported by most modern web services.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.
HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. If the client or server does not support HTTP/2, the client will automatically attempt to use an earlier version of the protocol.
HTTP_VERSION_30	The client should use the HTTP/3 protocol standard defined in RFC 9114. This protocol version is similar to earlier versions of the protocol in terms the request methods and server responses, but the underlying transport mechanism is significantly different. Instead of using TCP, this version of the protocol uses multiplexed UDP connections designed to reduce the overhead of TCP while maintaining reliability. If the client or server does not support HTTP/3, the client will automatically attempt to use an earlier version of the protocol.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

If the HTTP_PROXY_WINDOWS proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related parameters will be ignored

and the function will establish a connection directly to the server.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event occurs when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

Most applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support

the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return INVALID_CLIENT.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

The handle returned by this function is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **HttpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncConnect, HttpConnect, HttpDisconnect, HttpInitialize, HttpProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncPutData Function

```
UINT WINAPI HttpAsyncPutData(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPVOID lpvBuffer,
   DWORD dwLength,
   DWORD dwReserved,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Copies the contents of the specified buffer to a file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpvBuffer

The data that will be copied to the server and stored in the specified file.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPutData** function is used to upload the contents of a local buffer to the server. This function is similar to the **HttpPutData** function, however it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent to the server.

Because this function works asynchronously, it is important that the memory allocated for the buffer is

not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, ensure that your code does not return from the function while the data is being uploaded. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the transfer has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the ***lpvBuffer*** parameter on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

If the ***lpvBuffer*** parameter is pointing to a Unicode string, it is important to note that the value of the ***dwLength*** parameter should specify the number of bytes, not the number of characters. When using UTF-16, each character is two bytes long and therefore the length of the buffer is effectively double the length of the string. Because Unicode strings can contain null characters, you must also set the current file type to FILE_TYPE_IMAGE prior to calling this function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetData, HttpAsyncGetFile, HttpAsyncPutFile, HttpEventProc, HttpPutData, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncPutFile Function

```
UINT WINAPI HttpAsyncPutFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Uploads the specified file from the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error

information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPutFile** function is used to upload the contents of a local file to the server. This function is similar to the **HttpPutFile** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetData, HttpAsyncGetFile, HttpAsyncPutData, HttpEventProc, HttpPutFile, HttpTaskDone, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncPutFileEx Function

```
UINT WINAPI HttpAsyncPutFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Uploads the specified file from the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique

asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPutFileEx** function is used to upload the contents of a local file to the server. This function is similar to the **HttpPutFileEx** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAsyncGetFileEx, HttpEventProc, HttpGetFileEx, HttpPutFileEx, HttpTaskDone, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAsyncSubmitForm Function

```
UINT WINAPI HttpAsyncSubmitForm(
   HCLIENT hClient,
   HFORM hForm,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Submit the specified form to the server for processing and return the response to the caller.

Parameters

hClient

Handle to the client session.

hForm

Handle to the virtual form which contains the data to be submitted to the server.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the lpvBuffer parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_SUBMIT_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_SUBMIT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncSubmitForm** function is used to submit form data to a script that executes on the server and then copy the output from that script into a local buffer. This function is similar to the **HttpSubmitForm** function, however it submits the data using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent and the response from the server is being returned to the caller.

Because this function works asynchronously, it is important that the memory allocated for the *lpvResult* parameter will not be released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate the buffer on the heap and free that memory after the task has completed and the data is no longer needed.

Do not call the **HttpDestroyForm** function to release the memory allocated for the virtual form until the background task completes. Doing so may result in unpredictable behavior and could cause your application to terminate abnormally.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

Example

```
HFORM hForm = INVALID_FORM;
HGLOBAL hgblResult = (HGLOBAL)NULL;
DWORD cbResult = 0;
INT nResult = 0;
UINT nTaskId;

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);

if (hForm == INVALID_FORM)
    return;
```

```
HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nTaskId = HttpAsyncSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    // Get a pointer to the data returned by the server
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer now points to data returned by the server after
    // the form data was submitted

    // Release the memory allocated for the buffer
    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}

// Release the memory allocated for the virtual form
HttpDestroyForm(hForm);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddFormField, HttpAddFormFile, HttpClearForm, HttpCreateForm, HttpDeleteFormField, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAttachThread Function

```
DWORD WINAPI HttpAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **HttpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **HttpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **HttpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **HttpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **HttpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpConnect, HttpDisconnect, HttpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAuthenticate Function

```
INT WINAPI HttpAuthenticate(
   HCLIENT hClient,
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

Parameters

hClient

Handle to the client session.

nAuthType

An unsigned integer value which specifies the method to be used when authenticating the client. The following values are supported:

Value	Description
HTTP_AUTH_NONE	No client authentication should be performed. The ***lpszUserName*** and ***lpszPassword*** parameters are ignored and current authentication settings are cleared.
HTTP_AUTH_BASIC	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error. This function will not automatically refresh an expired token.

lpszUserName

A null-terminated string which specifies the user name used to authenticate the client session. This parameter may be NULL or an empty string if a user name is not required for the specified authentication type. If the authentication type is HTTP_AUTH_BEARER, this parameter is ignored.

lpszPassword

A null-terminated string that specifies the password used to authenticate the client session. If the authentication type is HTTP_AUTH_BEARER, this value must be the bearer token issued by the service provider.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function will set the **`Authorization`** request header for the client session using the credentials

provided by the caller. This function will always override any custom **Authorization** header value that may have been previously set using the **HttpSetRequestHeader** function.

If both the *lpszUserName* and *lpszPassword* parameters are NULL pointers or specify zero-length strings, the current authentication type will always be set to HTTP_AUTH_NONE regardless of the value of the *nAuthType* parameter. This effectively clears the current user credentials for the client session.

If the web service requires OAuth 2.0 authentication, it is recommended you use the **HttpSetBearerToken** function to specify the access token.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnect, HttpSetBearerToken, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCancel Function

```
INT WINAPI HttpCancel(
   HCLIENT hClient
);
```

The **HttpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When the **HttpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCancelQueuedFile Function

```
BOOL WINAPI HttpCancelQueuedFile(
    HQUEUE hQueue,
    DWORD dwFileId
);
```

The **HttpCancelQueuedFile** function cancels the queued file transfer.

Parameters

hQueue

A handle to a file transfer queue.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. If this value is zero, the function will cancel the current file transfer in progress. If queue transfers have been suspended, this parameter cannot be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

When this function is called, the queued file transfer may not immediately stop. An internal flag is set which causes the file transfer to exit with an error and the queue manager will begin processing the next file in the queue. If the queue is in an idle state, this function will fail.

It is permitted to call **HttpCancelQueuedFile** from within a queue event handler to cancel the current file transfer. In this case, the *dwFileId* parameter should be zero. If you specify a file which has not been transferred, it will be flagged as canceled and skipped by the queue manager when processing the queue. If you specify a file which has already been processed, this function will fail.

A canceled file transfer is also considered a failed transfer. When you call **HttpGetQueueStatus** after a queued file transfer is canceled, the *dwFailedFiles* member of the HTTPQUEUESTATUS structure will be incremented to reflect this change. You can determine the status of an individual file transfer by calling the **HttpGetQueuedFile** function and checking the value of the *dwQueueFlags* member of the HTTPQUEUEDFILE structure.

The **HttpResetQueue** function can be used to reset the state of previously canceled transfers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCancel, HttpGetQueuedFile, HttpGetQueueStatus, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpSuspendQueue, HttpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpClearForm Function

```
INT WINAPI HttpClearForm(
    HFORM hForm
);
```

The **HttpClearForm** function clears the specified form, removing all fields.

Parameters

hForm

Handle to the virtual form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpAddFormField, HttpAddFormFile, HttpCreateForm, HttpDeleteFormField, HttpDestroyForm, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpClearHeaders Function

```
INT WINAPI HttpClearHeaders(
    HCLIENT hClient
);
```

The **HttpClearHeaders** function resets the current request and response headers to their default values.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, it will return HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpClearHeaders** function removes all custom header values for the current client session and clears the response headers for the previous request. This function is useful when using persistent connections and you want to ensure that any custom header values from the previous request will not be included with subsequent requests made during the same session. Because this function clears all of the request headers for the client session, this means it will also remove any cookies which have been set using the **HttpSetCookie** function.

This function will clear any authentication credentials or tokens specified for the current client session. If the next request you make requires authentication, you must provide those credentials again using the **HttpAuthenticate** or **HttpSetBearerToken** function.

It is not necessary to use this function if the **HTTP_OPTION_KEEPALIVE** option was not specified when the connection was established. The headers for the previous request are automatically cleared when a new connection is established.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpAuthenticate, HttpConnect, HttpGetCookie, HttpGetResponseHeader, HttpSetBearerToken, HttpSetCookie, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpClearQueue Function

```
BOOL WINAPI HttpClearQueue(
    HQUEUE hQueue
);
```

The **HttpClearQueue** function removes all files from the queue.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function can only be called when the queue is in an idle state. An error will be returned if the function is called while the queue manager is paused or actively transferring files in the queue. To determine the current state of the queue, call the **HttpGetQueueStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCancelQueuedFile, HttpGetQueueStatus, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpStopQueue, HttpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCloseFile Function

```
INT WINAPI HttpCloseFile(
   HCLIENT hClient
);
```

The **HttpCloseFile** function flushes the internal client buffers and closes the previously opened file on the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the file is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the function returns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCommand, HttpGetData, HttpGetFile, HttpOpenFile, HttpPutData, HttpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCommand Function

```
INT WINAPI HttpCommand(
   HCLIENT hClient,
   LPCTSTR lpszCommand,
   LPCTSTR lpszResource,
   LPBYTE lpParameter,
   DWORD cbParameter,
   DWORD dwReserved
);
```

The **HttpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for extended commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

A null-terminated string which specifies the command to be executed by the server. The following table lists the standard commands recognized by most HTTP servers. Other commands may also be used, such as those extensions used by WebDAV to edit and manage files on a server.

Command	Description
GET	Return the contents of the specified resource. This command is recognized by all servers.
HEAD	Return only header information for the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
POST	Post data to the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
PUT	Create or replace the specified resource on the server. This command is recognized by servers that support at least version 1.0 of the protocol. Not all servers support this command.
DELETE	Delete the specified resource from the server. This command is recognized by servers that support at least version 1.1 of the protocol. Not all servers support this command.

lpszResource

A null-terminated string which specifies the resource to be used with the command. This can be the name of a file, an executable script or any other valid resource name recognized by the server. Resource names must be absolute and include the complete path to the resource.

lpParameter

A pointer to a byte array which contains data that is to be passed to the command as one or more parameters. Typically this is used to pass additional information to a script that executes on the server. The data is encoded according to the encoding type specified for the client session. If the resource does not require any parameters, this value should be NULL.

cbParameter

Specifies the number of bytes stored in the parameter buffer. If the resource does not require any

parameters, this value should be zero.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the result code returned by the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Not all servers support all of the listed commands, and some commands may require specific changes to the server configuration. In particular, the PUT and DELETE commands typically require that configuration changes be made by the site administrator. All servers will support the use of the GET command, and all servers that support at least version 1.0 of the protocol will support the POST command.

If the *lpszResource* parameter specifies a path that contains reserved or restricted characters, such as a space, it will automatically be URL encoded by the library.

It is permissible to include a query string in the resource name specified by the *lpszResource* parameter. Query strings begin with a question mark, and then are followed by one or more name/value pairs separated by an equal sign. For example, the following resource includes a query string:

/cgi-bin/test.cgi?field1=value1&field2=value2

In this case, the query string is "?field1=value1&field2=value2". If the query string contains reserved or restricted characters, such as spaces, then it will be automatically URL encoded prior to being sent to the server. If additional resource data is specified in the *lpParameter* argument along with a query string in the resource name, the action taken by the library depends on the command being sent. If the command is a POST or PUT command, then query string is included with command request to the server and the parameter data is sent separately. For example, if the POST command was used, the script running on the server would see that both query data and form data has been provided to it. However, if any other command is specified, the parameter data is simply appended to the query string.

The *lpParameter* argument is used to pass additional information to the server when a resource is requested. This is most commonly used to provide information to scripts, similar to how arguments are used when executing a program from the command line. Unless the POST command is being executed, the data in the buffer will automatically be encoded using the current encoding mechanism specified for the client. By default, the data is URL encoded, which means that any spaces and non-printable characters are converted to printable characters before submitted to the server. The type of encoding that is performed can be set by calling the **HttpSetEncodingType** function. Although the default encoding is appropriate for most applications, those that submit XML formatted data may need to change the encoding type.

Only one request may be in progress at one time for each client session. Use the **HttpCloseFile** function to terminate the request after all of the data has been read from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCloseFile, HttpCreateFile, HttpGetData, HttpGetFile, HttpGetResultCode, HttpGetResultString, HttpOpenFile, HttpPostData, HttpPutData, HttpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpConnect Function

```
HCLIENT WINAPI HttpConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion,
   LPSECURITYCREDENTIALS lpCredentials
);
```

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration

	can be changed in the system settings or control panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the client will default to using TLS 1.2 or

	later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create a version value, or the following values can be used:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.
HTTP_VERSION_09	The client should use the original one-line protocol which includes no version number and no request header block. This version has been deprecated and should only be used with legacy servers which do not support the protocol standard. This version of the protocol does not support virtual hosts and

	is not supported by most modern web services.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.
HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. If the client or server does not support HTTP/2, the client will automatically attempt to use an earlier version of the protocol.
HTTP_VERSION_30	The client should use the HTTP/3 protocol standard defined in RFC 9114. This protocol version is similar to earlier versions of the protocol in terms the request methods and server responses, but the underlying transport mechanism is significantly different. Instead of using TCP, this version of the protocol uses multiplexed UDP connections designed to reduce the overhead of TCP while maintaining reliability. If the client or server does not support HTTP/3, the client will automatically attempt to use an earlier version of the protocol.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Most applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support

the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically attempt to reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Under some circumstances, a server may forcibly close or abort a connection if the request fails. For example, this can happen when a malformed request is submitted to the server or a server-side error occurs. When this happens, the application will need to disconnect from the server and establish a new connection, even if the keep-alive option has been specified.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return INVALID_CLIENT.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

The handle returned by this function is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpDisconnect, HttpInitialize, HttpProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpConnectUrl Function

```
HCLIENT WINAPI HttpConnectUrl(
    LPCTSTR lpszUrl,
    UINT nTimeout,
    DWORD dwOptions
);
```

The **HttpConnectUrl** function establishes a connection with the specified server using a URL.

Parameters

lpszUrl

A null-terminated string which specifies the URL for the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the

	content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. If this option is not specified, the connection will use HTTP/1.1 as the default protocol version.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the client will default to using TLS 1.2 or later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use

	of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpConnectUrl** function uses an HTTP URL to establish a connection with a server. The URL must be in the following format:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified. The **HttpValidateUrl** function can be used to verify that a URL is valid prior to calling this function.

The **HttpConnectUrl** function is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection functions. If you only need to upload or download a file using a URL, then refer to the **HttpUploadFile** and **HttpDownloadFile** functions.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **HttpConnectUrl** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Most applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

If your application specifies the HTTP_OPTION_HTTP2 option, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of Windows do not support the features required for a secure HTTP/2 connection. If the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return INVALID_CLIENT.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

The handle returned by this function is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
HCLIENT hClient;
LPCTSTR lpszUrl = _T("https://www.example.com/");

// Connect to the site specified by the URL
hClient = HttpConnectUrl(lpszUrl, HTTP_TIMEOUT, HTTP_OPTION_DEFAULT);

if (hClient == INVALID_CLIENT)
```

```
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION | MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnect, HttpDisconnect, HttpDownloadFile, HttpUploadFile, HttpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCreateFile Function

```
INT WINAPI HttpCreateFile(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   DWORD dwLength,
   DWORD dwOffset
);
```

The **HttpCreateFile** function creates the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

Points to a string that specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

dwLength

Specifies the length of the file that will be created on the server. This value must be greater than zero.

dwOffset

Specifies a byte offset in the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to create the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpCreateFile** function uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this function is successful, the client should then use the **HttpWrite** function to send the contents of the file to the server. Once all of the data has been written, the **HttpCloseFile** function should be called to close the file and complete the operation. Note that this function is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **HttpWrite** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file in a single function call, use the **HttpPutData** function instead of calling **HttpCreateFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCloseFile, HttpOpenFile, HttpPutData, HttpPutFile, HttpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCreateFileEx Function

```
INT WINAPI HttpCreateFileEx(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   ULARGE_INTEGER uiLength,
   ULARGE_INTEGER uiOffset
);
```

The **HttpCreateFileEx** function creates the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

Points to a string that specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

uiLength

Specifies the length of the file that will be created on the server. This value must be greater than zero.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to create the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpCreateFileEx** function uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this function is successful, the client should then use the **HttpWrite** function to send the contents of the file to the server. Once all of the data has been written, the **HttpCloseFile** function should be called to close the file and complete the operation. Note that this function is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **HttpWrite** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file in a single function call, use the **HttpPutData** function instead of calling **HttpCreateFileEx**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCloseFile, HttpOpenFile, HttpPutData, HttpPutFile, HttpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCreateForm Function

```
HFORM WINAPI HttpCreateForm(
   LPCTSTR lpszAction,
   UINT nFormMethod,
   UINT nFormType,
   DWORD dwReserved
);
```

The **HttpCreateForm** function creates a new form and returns a handle for use with the other form-related functions.

Parameters

lpszAction

A null-terminated string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. This parameter may be one of the following values:

Value	Description
HTTP_METHOD_DEFAULT	The form data should be submitted using the default method, using the GET command.
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. This parameter may be one of the following values:

Value	Description
HTTP_FORM_DEFAULT	The form data should be submitted using the default encoding method.
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is a handle to the virtual form. If the function fails, the return value is INVALID_FORM. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpCreateForm** function is used to create a new form that will be populated with values and then submitted to the server for processing. When the form is no longer needed, it should be destroyed using the **HttpDestroyForm** function.

Example

```
HFORM hForm = INVALID_FORM;
HGLOBAL hgblResult = (HGLOBAL)NULL;
DWORD cbResult = 0;
INT nResult = 0;

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);

if (hForm == INVALID_FORM)
    return;

HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nResult = HttpSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);
HttpDestroyForm(hForm);

if (hgblResult != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer points to data returned by the server after the form
    // data was submitted

    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddFormField, HttpAddFormFile, HttpClearForm, HttpDeleteFormField, HttpDestroyForm, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCreateQueue Function

```
HQUEUE WINAPI HttpCreateQueue(
    UINT nMaxFiles,
    UINT nTimeout,
    DWORD dwOptions
);
```

The **HttpCreateQueue** function creates new file transfer queue.

Parameters

nMaxFiles

An unsigned integer which specifies the maximum number of files which can be added to the queue. The constant INFINITE can be used to specify that there is no fixed limit to the number of files in the queue.

nTimeout

An unsigned integer which specifies the default timeout for all queued file transfers. If this value is zero, a reasonable default timeout period will be used. This timeout period is used when a file is added to the queue without providing a timeout period for that specific file transfer. If a timeout period is specified for a particular file, it will override this value.

dwOptions

An unsigned integer value which specifies one or more default options for all queued files. This parameter is constructed by using a bitwise operator and can be any of the options used with the **HttpConnect** function. If transfer options are specified for a particular file, it will override this value. The most common options are:

Value	Description
HTTP_OPTION_DEFAULT	Default options should be used for queued file transfers. All standard headers will be included with the request and redirected resources will return an error, allowing the application to determine if it should request the resource from the new location or inform the user the resource location has changed.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect

	using an earlier version of the protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.

Return Value

If the function succeeds, the return value is a handle to the queue which can be used with subsequent function calls. If the function fails, the return value is INVALID_QUEUE. To get extended error information, call **HttpGetLastError**.

Remarks

If the *nMaxFiles* parameter is INFINITE, memory will be dynamically allocated on the process heap with no limit to the number of files which can be queued. If there is a logic error which causes the application to recursively add files to the queue, or repeatedly queue the same file, this can result in virtual memory being exhausted for the process. You can avoid this by specifying a reasonable maximum queue size, which will cause the **HttpAddQueuedFile** function to fail if that limit is exceeded.

The memory allocated for the queue will be released when the **HttpDeleteQueue** function is called.

Example

```
// Create a new queue
HQUEUE hQueue = HttpCreateQueue(INFINITE, HTTP_TIMEOUT, HTTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = HttpAddQueuedFile(
                     hQueue,
                     lpszLocalFile,
                     lpszRemoteFile,
                     HTTP_OPTION_DEFAULT,
                     HTTP_TIMEOUT);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (HttpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
```

```
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
            }
        }
    }

    // Start the queued file transfers and wait for it to complete
    if (HttpStartQueue(hQueue, HTTP_QUEUE_ALL, 0, NULL, 0))
    {
        HttpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
        HttpStopQueue(hQueue);
    }

    // Remove all files from the queue
    HttpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpAddQueuedFile, HttpDeleteQueue, HttpClearQueue, HttpStartQueue, HttpStopQueue, HttpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCreateSecurityCredentials Function

```
BOOL WINAPI HttpCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **HttpCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **HttpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **HttpConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
HttpCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                              0,
                              NULL,
                              NULL,
                              lpszCertStore,
                              lpszCertName,
                              NULL,
                              &lpSecCred);

hClient = HttpConnect(lpszHostName,
                      HTTP_PORT_SECURE,
                      HTTP_TIMEOUT,
                      HTTP_OPTION_SECURE | HTTP_OPTION_KEEPALIVE,
                      HTTP_VERSION_11,
                      lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnect, HttpDeleteSecurityCredentials, HttpGetSecurityInformation, HttpProxyConnect, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteFile Function

```
INT WINAPI HttpDeleteFile(
   HCLIENT hClient,
   LPCTSTR lpszFileName
);
```

The **HttpDeleteFile** function deletes the specified file from the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function uses the DELETE command to delete the specified file from the server. The server must be configured to support this command, and client must have the appropriate permission to delete the file, or an error will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetFile, HttpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteFormField Function

```
INT WINAPI HttpDeleteFormField(
   HFORM hForm,
   LPCTSTR lpszFieldName
);
```

The **HttpDeleteFormField** function deletes the specified field from the form.

Parameters

hForm

Handle to the virtual form.

lpszFieldName

Points to a string that specifies the name of the field to delete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddFormField, HttpAddFormFile, HttpClearForm, HttpCreateForm, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteHeaders Function

```
BOOL WINAPI HttpDeleteHeaders(
    HCLIENT hClient,
    UINT nHeaderType
);
```

Delete all of the response or request headers for the current session.

Parameters

hClient

Handle to the client session.

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that have been set for the next request sent to the server. This will clear all of the header values that were set using the **HttpSetRequestHeader** function.
HTTP_HEADERS_RESPONSE	Delete all of the headers that were set in response to the previous request. A call to the **HttpGetResponseHeader** function to obtain a specific header value will fail after this function returns.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpDeleteHeaders** function will release the memory allocated for the request or response headers for the current session. This function is typically used to clear all of the current request headers so that the application may create a new set of headers when a persistent connection is being used. The memory allocated for the request and response headers is normally released when the session handle is closed by calling the **HttpDisconnect** function.

Whenever a request for a resource is sent to the server, the response headers from the previous request are automatically cleared. It is not necessary for an application to call the **HttpDeleteHeaders** function to delete the response headers prior to each request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetFirstHeader, HttpGetNextHeader, HttpGetResponseHeader, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteQueue Function

```
BOOL WINAPI HttpDeleteQueue(
    HQUEUE hQueue
);
```

The **HttpDeleteQueue** function deletes the specified file transfer queue, releasing all memory allocated for the queued files.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function can only be called when the queue is in an idle state. If the queue manager is in the process of uploading or downloading files, the function will fail and return a value of zero. You can determine the current state of the queue by calling the **HttpGetQueueStatus** function. It is not necessary to explicitly clear the queue prior to calling this function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCreateQueue, HttpGetQueueStatus, HttpCancelQueuedFile, HttpClearQueue, HttpStartQueue, HttpStopQueue, HttpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteSecurityCredentials Function

```
VOID WINAPI HttpDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **HttpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)
    HttpDeleteSecurityCredentials(&lpSecCred);

HttpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCreateSecurityCredentials, HttpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDestroyForm Function

```
INT WINAPI HttpDestroyForm(
    HFORM hForm
);
```

The **HttpDestroyForm** function destroys the specified form, releasing the memory allocated for it.

Parameters

hForm

Handle to the form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpAddFormField, HttpAddFormFile, HttpClearForm, HttpCreateForm, HttpDeleteFormField, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDisableEvents Function

```
INT WINAPI HttpDisableEvents(
    HCLIENT hClient
);
```

The **HttpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpEnableEvents, HttpFreezeEvents, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDisableTrace Function

```
BOOL WINAPI HttpDisableTrace();
```

The **HttpDisableTrace** function disables the logging of network function calls.

Parameters

None.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDisconnect Function

```
INT WINAPI HttpDisconnect(
   HCLIENT hClient
);
```

The **HttpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpConnect, HttpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDownloadFile Function

```
BOOL WINAPI HttpDownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPHTTPTRANSFERSTATUS lpStatus
   HTTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **HttpDownloadFile** function downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.

HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpDownloadFile** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then download the file to the local system. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return a value of zero.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpDownloadFile** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpGetFile** to download the file.

Example

```
HTTPTRANSFERSTATUS httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile = _T("http://www.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = HttpDownloadFile(lpszLocalFile,
                           lpszRemoteFile,
                           HTTP_OPTION_DEFAULT,
                           &httpStatus,
                           NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEventProc, HttpGetFile, HttpGetFileZoneInfo, HttpGetTransferStatus, HttpSetFileZoneInfo, HttpSubmitRequest, HttpUploadFile, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDownloadFileEx Function

```
BOOL WINAPI HttpDownloadFileEx(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPHTTPTRANSFERSTATUSEX lpStatus
   HTTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **HttpDownloadFileEx** function downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.

HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpStatus

A pointer to an HTTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpDownloadFileEx** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then download the file to the local system. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return a value of zero.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpDownloadFileEx** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpGetFile** to download the file.

Example

```
HTTPTRANSFERSTATUSEX httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile = _T("http://www.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = HttpDownloadFileEx(lpszLocalFile,
                             lpszRemoteFile,
                             HTTP_OPTION_DEFAULT,
                             &httpStatus,
                             NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEventProc, HttpGetFileEx, HttpGetTransferStatusEx, HttpUploadFileEx, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnableCompression Function

```
INT WINAPI HttpEnableCompression(
    HCLIENT hClient,
    BOOL bEnable
);
```

The **HttpEnableCompression** function enables or disables support for data compression.

Parameters

hClient

Handle to the client session.

bEnable

A boolean value which specifies if data compression should be enabled or disabled. A non-zero value enables compression, while a value of zero will disable compression.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpEnableCompression** function is used to indicate to the server whether or not it is acceptable to compress the data that is returned to the client. If compression is enabled, the client will advertise that it will accept compressed data by setting the **Accept-Encoding** request header. The server will decide whether a resource being requested can be compressed. If the data is compressed, the library will automatically expand the data before returning it to the caller.

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. Disabling compression informs the server that the client is not willing to accept compressed data; this is the default.

If the **HttpSetRequestHeader** function is used to explicitly set the **Accept-Encoding** header to request compressed data and compression is not enabled, the library will not attempt to automatically expand the data returned by the server. In this case, the raw compressed data will be returned to the caller and the application is responsible for processing it. This behavior is by design to maintain backwards compatibility with previous versions of the library that did not have internal support for compression.

To determine if the server compressed the data returned to the client, use the **HttpGetResponseHeader** function to get the value of the **Content-Encoding** header. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

Enabling compression is only meaningful when downloading files from a server that supports file compression. It has no effect on file uploads.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetData, HttpGetFile, HttpGetFileEx, HttpGetResponseHeader, HttpGetText, HttpSetRequestHeader

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnableEvents Function

```
INT WINAPI HttpEnableEvents(
    HCLIENT hClient,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **HtpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **HttpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the ***hEventWnd*** parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event occurs when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpDisableEvents, HttpFreezeEvents, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnableTrace Function

```
BOOL WINAPI HttpEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **HttpEnableTrace** function enables the logging of Windows Sockets function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

When logging is enabled for HTTP/2 and HTTP/3 client sessions, it is normal to see HTTP/1.1 in the

request and server response. They are binary protocols and the request and response header blocks emulate a standard HTTP/1.1 text response for backwards compatibility. Applications running on the server should work in the same way regardless of which protocol version is selected, however it is possible to check the server environment to determine which version of HTTP was used with the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnumQueuedFiles Function

```
LONG WINAPI HttpEnumQueuedFiles(
   HQUEUE hQueue,
   LPCTSTR lpszFileMask,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPDWORD lpFileList,
   LONG nMaxFiles
);
```

The **HttpEnumQueuedFiles** function returns a list of files in the current transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

lpszFileMask

A null-terminated string which specifies a file name which can include wildcards. Only those files which match this value will be enumerated. The character '?' will match against any single character, and '*' will match any number of characters. If this parameter is NULL or points to an empty string, all eligible files in the queue will be matched.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. Only those files which match the queue mode will be enumerated. This is a bitmask which may be one or more of the following values:

Constant	Description
HTTP_QUEUE_ALL	Match the file name to all files in the queue.
HTTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
HTTP_QUEUE_UPLOAD	Match file names which are queued for upload.
HTTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the enumeration. If this value is zero, no files will be excluded; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_QUEUE_FLAG_NONE	No files should be excluded.
HTTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
HTTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.

HTTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **HttpCancelQueuedFile** function has been called and a queued file is in the process of being uploaded or downloaded.

lpFileList

A pointer to an array of unsigned integer values which will contain the unique file identifiers for each matching file in the queue. This parameter must specify an array large enough to store all of the file identifiers, otherwise the function will fail with a ST_ERROR_BUFFER_TOO_SMALL error. If this parameter is NULL, the function will return the number of matching files.

nMaxFiles

An integer value which specifies the maximum number of file identifiers which can be copied into the *lpFileList* array. If the *lpFileList* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of matching files. If the queue is empty or there are no matching files, this function will return zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function populates an array of unique file identifiers which can be used to obtain information about the files in the current queue. The values returned in the *lpFileList* array can be used in conjunction with the **HttpGetQueuedFile** function. This provides an alternative to using the **HttpGetFirstQueuedFile** and **HttpGetNextQueuedFile** functions.

The application should not make any assumptions about the value of the file identifiers returned by this function. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

Example

```
// Get the total number of files in the queue
LONG nFiles = HttpEnumQueuedFiles(hQueue, NULL, HTTP_QUEUE_ALL,
HTTP_QUEUE_FLAG_NONE, NULL, 0)

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];

    nFiles = HttpEnumQueuedFiles(hQueue, NULL, HTTP_QUEUE_ALL,
HTTP_QUEUE_FLAG_NONE, pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
        HTTPQUEUEDFILE queuedFile;

        if (HttpGetQueuedFile(hQueue, pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
```

```
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpFindQueuedFile, HttpGetFirstQueuedFile, HttpGetNextQueuedFile, HttpGetQueuedFile, HttpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnumTasks Function

```
INT WINAPI HttpEnumTasks(
   UINT *lpTasks,
   INT nMaxTasks,
   DWORD dwOptions
);
```

Return a list of active, suspended or finished asynchronous tasks.

Parameters

lpTasks

A pointer to an array of unsigned integer values that will contain unique task identifiers when the function returns. If this parameter is NULL, the function will return the number of tasks.

nMaxTasks

An integer value that specifies the maximum number of task identifiers that may be copied into the ***lpTasks*** array. If the ***lpTasks*** parameter is NULL, this value must be zero.

dwOptions

An unsigned integer that specifies the type of asynchronous tasks that may be returned by this function. It may be a combination of the following values:

Value	Description
HTTP_TASK_DEFAULT	The list of asynchronous task IDs should include both active and suspended tasks. This option is the same as specifying both the HTTP_TASK_ACTIVE and HTTP_TASK_SUSPENDED options.
HTTP_TASK_ACTIVE	The list of asynchronous task IDs should include those tasks which are currently active. An active task represents a background connection to a server that is in the process of performing the requested action, such as uploading or downloading a file.
HTTP_TASK_SUSPENDED	The list of asynchronous task IDs should include those tasks which have been suspended. A suspended task represents a background connection that has been established, but the worker thread is not scheduled for execution.
HTTP_TASK_FINISHED	The list of asynchronous task IDs should include those tasks which have completed recently.

Return Value

If the function is successful, the return value is the number of task identifiers copied into the provided array. If there are no tasks which match the requested criteria, the return value is zero. A return value of HTTP_ERROR indicates an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpEnumTasks** function can be used to obtain a list of numeric identifiers that represent the asynchronous tasks that have been started or those that have completed. These task IDs are used by other functions to reference the background worker thread that has been created and obtain status

information for the task. For example, the **HttpTaskDone** function can be used to determine if a particular task has completed, and the **HttpTaskWait** function can be used to wait for a task to complete and return an error status code if the background operation failed.

There is an internal limit of 128 asynchronous tasks per process that may be active at any one time. When a task completes, the status information about that task is maintained for period of time after the task has completed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskDone, HttpTaskResume, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEventProc Function

```
VOID CALLBACK HttpEventProc(
   HCLIENT hClient,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **HttpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **HttpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **HttpRegisterEvent** function. This callback function is also used by asynchronous tasks to notify the application when the task has started and completed. The **HttpEventProc** function is a placeholder for the application-defined function name.

If the callback function is invoked by an asynchronous task, it will execute in the context of the worker thread that is managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function for an asynchronous task. Controls should only be modified by the same UI thread that created their window. One common approach to resolve this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpDisableEvents, HttpEnableEvents, HttpFreezeEvents, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpFindQueuedFile Function

```
DWORD WINAPI HttpFindQueuedFile(
    HQUEUE hQueue,
    LPCTSTR lpszFileName,
    DWORD dwQueueMode,
    DWORD dwExcludeFiles
);
```

The **HttpFindQueuedFile** function finds a file in the transfer queue which matches the search criteria.

Parameters

hQueue

A handle to a file transfer queue.

lpszFileName

A null-terminated string which specifies the name of the file to search for. The string may contain the wildcard character '?' to match against any single character, and '*' will match any number of characters. This parameter cannot be NULL or point to an empty string.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. This is a bitmask which may be one or more of the following values:

Constant	Description
HTTP_QUEUE_ALL	Match the file name to all files in the queue.
HTTP_QUEUE_DOWNLOAD	Match file names which are queued for download.
HTTP_QUEUE_UPLOAD	Match file names which are queued for upload.
HTTP_QUEUE_LOCAL	Match against the local file name instead of the remote file name.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_QUEUE_FLAG_NONE	No files should be excluded from the search.
HTTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
HTTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
HTTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the

	HttpCancelQueuedFile function has been called and a queued file is in the process of being uploaded or downloaded.

Return Value

If the function succeeds, the return value is a unique identifier for the file. To obtain information about the file, call the **HttpGetQueuedFile** function. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns a unique file identifier for the first file which matches the specified file name in the queue. If the *lpszFileName* parameter includes wildcard characters, this function will return the first file in the queue which matches the name. File name matches are not case-sensitive, even when matching against remote file names on servers which use case-sensitive naming, such as UNIX based servers.

To perform more complex searches based on the file name, or to find multiple files, you can use the **HttpGetFirstQueuedFile** and **HttpGetNextQueuedFile** functions to iterate through all queued files.

Example

```
// Find a local file in the download queue
DWORD dwFileId = HttpFindQueuedFile(
                    hQueue,
                    lpszFileName,
                    HTTP_QUEUE_DOWNLOAD | HTTP_QUEUE_LOCAL,
                    HTTP_QUEUE_FLAG_NONE);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (HttpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                break;
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpEnumQueuedFiles, HttpGetFirstQueuedFile, HttpGetNextQueuedFile, HttpGetQueuedFile, HttpRemoveQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpFreezeEvents Function

```
INT WINAPI HttpFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **HttpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpDisableEvents, HttpEnableEvents, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetBearerToken Function

```
INT WINAPI HttpGetBearerToken(
   HCLIENT hClient,
   LPTSTR lpszBearerToken,
   INT nMaxLength
);
```

The **HttpGetBearerToken** function returns the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

hClient

Handle to the client session.

lpszBearerToken

A null-terminated string buffer which will contain the bearer token when the function returns. The string will be null-terminated and the buffer must be large enough to accommodate the entire bearer token or the function will fail. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the length of the bearer token string. A return value of zero indicates that no bearer token has been specified. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns the bearer token which was previously set by a call to **HttpSetBearerToken**. Bearer tokens can be very long strings and do not contain any human readable information. For Google services, these tokens are usually about 180 characters in length. For Microsoft services, their bearer tokens are typically 1,200 characters in length. It is recommended that you provide a buffer size of at least 2,000 characters.

Your application should not store the bearer tokens provided by a web service. These tokens are short-lived and typically only valid for about an hour. If the token has expired, the authorization to access the resource will fail and it must be refreshed. The refresh tokens used to acquire a new bearer token should be stored and they are typically valid for a period of months.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticate, HttpSetBearerToken

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetContentLength Function

```
BOOL WINAPI HttpGetContentLength(
    HCLIENT hClient,
    LPDWORD lpdwLength
);
```

The **HttpGetContentLength** function returns the size of the current resource in bytes.

Parameters

hClient

Handle to the client session.

lpdwLength

A pointer to a unsigned integer which will contain the content length for the current resource. This value will be initialized to zero and updated with the length when the function returns.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

When this function is called after a GET or POST request has completed, the ***lpdwLength*** parameter will contain the total number of bytes returned by the server. If this function is called while a request is being processed (for example, from within an event handler), it will return the value of the Content-Length response header provided by the server. If it's possible the server will return more than 4GiB of data to the client, the **HttpGetContentLengthEx** function should be used. That version of the function accepts an unsigned 64-bit integer and will prevent the content length from being truncated to a 32-bit value.

Most servers will only include a Content-Length header in the response for static resources like documents or image files. For other types of resources where the response payload is generated dynamically, such as server-side scripts, servers typically use "chunked" output and there is no Content-Length header. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. In this case, the content length can only be determined after the request has completed.

This function should only be used after a request has completed and will not return a value for dynamic resources as they are being retrieved from the server. To obtain the current number of bytes copied from within an event handler, use the **HttpGetTransferStatus** function.

If the content type for the current resource is textual, the content length reported by the server may be different than what is stored on the local system due to different end-of-line character conventions. For example, on Windows the end of a line of text is indicated by a carriage-return and linefeed pair, while on Linux the end-of-line is indicated by a single linefeed. Depending on the options specified when the request is made, text returned by the server may be normalized for the Windows platform.

This function does not use the HEAD command to request the size of a document on the server. It will only return the content length of a resource which has already been requested, typically after using functions such as **HttpGetText** or **HttpPostXml**. To ask the server for the size of document without downloading the contents of that document, use the **HttpGetFileSize** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetContentLengthEx, HttpGetContentType, HttpGetFileSize, HttpGetResponseHeader, HttpGetTransferStatus, HttpSetContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetContentLengthEx Function

```
BOOL WINAPI HttpGetContentLengthEx(
   HCLIENT hClient,
   ULARGE_INTEGER *lpLength
);
```

The **HttpGetContentLengthEx** function returns the size of the current resource in bytes.

Parameters

hClient

Handle to the client session.

lpLength

A pointer to a ULARGE_INTEGER union which will contain the content length for the current resource. This value will be initialized to zero and updated with the length when the function returns.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

When this function is called after a GET or POST request has completed, the ***lpLength*** parameter will contain the total number of bytes returned by the server. If this function is called while a request is being processed (for example, from within an event handler), it will return the value of the Content-Length response header provided by the server. This version of the function accepts an unsigned 64-bit integer as the ***lpLength*** parameter and is intended to support very large documents which exceed 4GiB.

Most servers will only include a Content-Length header in the response for static resources like documents or image files. For other types of resources where the response payload is generated dynamically, such as server-side scripts, servers typically use "chunked" output and there is no Content-Length header. In this case, the content length can only be determined after the request has completed.

This function should only be used after a request has completed and will not return a value for dynamic resources as they are being retrieved from the server. To obtain the current number of bytes copied from within an event handler, use the **HttpGetTransferStatusEx** function.

If the content type for the current resource is textual, the content length reported by the server may be different than what is stored on the local system due to different end-of-line character conventions. For example, on Windows the end of a line of text is indicated by a carriage-return and linefeed pair, while on Linux the end-of-line is indicated by a single linefeed. Depending on the options specified when the request is made, text returned by the server may be normalized for the Windows platform.

This function does not use the HEAD command to request the size of a document on the server. It will only return the content length of a resource which has already been requested, typically after using functions such as **HttpGetText** or **HttpPostXml**. To ask the server for the size of document without downloading the contents of that document, use the **HttpGetFileSizeEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetContentLength, HttpGetContentType, HttpGetFileSizeEx, HttpGetResponseHeader, HttpGetTransferStatusEx, HttpSetContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetContentType Function

```
INT WINAPI HttpGetContentType(
   HCLIENT hClient,
   LPTSTR lpszContentType,
   INT nMaxLength
);
```

The **HttpGetContentType** function returns the content type for the current resource.

Parameters

hClient

Handle to the client session.

lpszContentType

A buffer which will contain the MIME content type of the resource returned by the web server. The buffer must be large enough to contain the entire content type string including the terminating null character. This parameter is optional and may be NULL if the information is not required.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer. If the ***lpszContentType*** parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the value indicates the type of resource returned by the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Value	Description
HTTP_CONTENT_UNKNOWN	The content type for the resource could not be determined. This can occur if the content type returned by the server does not match a known value, such as an experimental or non-standard content type.
HTTP_CONTENT_BINARY	Application specific binary data which doesn't explicitly fall into any of the other standard MIME types. The most common type for binary data is application/octet-stream which can represent any type of data. Other examples are application/pdf and application/zip.
HTTP_CONTENT_TEXT	Textual data which can include plain text, HTML documents, XML, comma-separated value (CSV) data or any other human readable text formats. Examples are text/plain and text/html.
HTTP_CONTENT_AUDIO	Audio data which can include MEPG audio and Windows Media formats. Examples are audio/mpeg, audio/mp4 and audio/x-ms-wma.
HTTP_CONTENT_IMAGE	Image data which can include GIF, JPEG and PNG image formats. Examples are image/gif and image/jpeg.
HTTP_CONTENT_MESSAGE	A content type which specifies the response includes other messages, typically email messages in a standard MIME format. This content type is not commonly used with web

	services.
HTTP_CONTENT_MULTIPART	A content type which includes multiple types of data in a single response. The payload will typically be encoded text and the contents will need to be decoded to extract any binary data included in the message.
HTTP_CONTENT_VIDEO	Video data which can include MP4 movies or Window Media format video. Examples are video/mp4 and video/x-ms-wmv.
HTTP_CONTENT_FONT	Font data which can include TrueType and OpenType fonts. Examples are font/tff and font/otf.
HTTP_CONTENT_MODEL	Model data for 3D objects and scenes, such as those used with the Virtual Reality Modeling Language (VRML). Examples are model/3mf and model/vrml.

Remarks

This function does not examine the contents of the payload returned by the server and only returns a value based on the Content-Type response header. If the web server does not recognize the data format of the resource it is returning, it should identify it as **application/octet-stream** and the content type would be returned as HTTP_CONTENT_BINARY. However, some servers incorrectly return unrecognized formats as **text/plain**, causing the payload to be identified as human-readable text rather than binary data.

Some servers will return a content type of **text/plain** for JSON responses and others will use the IANA standard type of **application/json**. To ensure consistency with these two content types, this function will always return MIME_CONTENT_TEXT for JSON responses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetContentLength, HttpGetResponseHeader, HttpSetContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCookie Function

```
BOOL WINAPI HttpGetCookie(
   HCLIENT hClient,
   LPCTSTR lpszCookieName,
   LPTSTR lpszCookieValue,
   INT nCookieValue,
   LPTSTR lpszCookiePath,
   INT nCookiePath,
   LPTSTR lpszCookieDomain,
   INT nCookieDomain,
   LPSYSTEMTIME lpCookieExpires,
   LPDWORD lpdwCookieFlags
);
```

The **HttpGetCookie** function returns information about a cookie set by the server.

Parameters

hClient

Handle to the client session.

lpszCookieName

A null-terminated string which specifies the name of the cookie to return information about.

lpszCookieValue

A null-terminated string buffer that will contain the value of the cookie. If this information is not required, a NULL pointer may be specified.

nCookieValue

The maximum number of characters that may be copied into the buffer specified by the *lpszCookieValue* parameter, including the terminating null character.

lpszCookiePath

A null-terminated string buffer that will contain the cookie path. If this information is not required, a NULL pointer may be specified.

nCookiePath

The maximum number of characters that may be copied into the buffer specified by the *lpszCookiePath* parameter, including the terminating null character.

lpszCookieDomain

A null-terminated string buffer that will contain the cookie domain. If this information is not required, a NULL pointer may be specified.

nCookieDomain

The maximum number of characters that may be copied into the buffer specified by the *lpszCookieDomain* parameter, including the terminating null character.

lpCookieExpires

A pointer to a SYSTEMTIME structure which specifies the date and time that the cookie expires. If this information is not required, a NULL pointer may be specified.

lpdwCookieFlags

One or more bit flags which specify status information about the cookie. A value of zero indicates that there are no special status flags for the cookie. This parameter may be NULL if the information

is not required. The following values are currently defined:

Value	Description
HTTP_COOKIE_SECURE	This flag specifies that the cookie should only be provided to the server if the connection is secure.
HTTP_COOKIE_SESSION	This flag specifies that the cookie should only be used for the current application session and should not be stored permanently on the local system.

Return Value

If the function succeeds, the return value is non-zero. If the specified cookie does not exist or function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The Hypertext Transfer Protocol uses special tokens called "cookies" to maintain persistent state information between requests for a resource. These cookies are exchanged between the client and server by setting specific header fields. When a server wants the client to use a cookie, it will include a header field named Set-Cookie in the response header when the client requests a resource. The client can then take this cookie and store it, either temporarily in memory or permanently in a file on the local system. The next time that the client requests a resource from that server, it can send the cookie back to the server by setting the Cookie header field. The **HttpGetCookie** function searches for a cookie set by the server in the Set-Cookie header field. The **HttpSetCookie** function creates or modifies the Cookie header field for the next resource requested by the client.

There are two general types of cookies that are used by servers. Session cookies exist only for the duration of the client session; they are stored in memory and not saved in any kind of permanent storage. When the client application terminates, session cookies are deleted and no longer used. Persistent cookies are stored on the local system and are used by the client until their expiration time. It is the responsibility of the client application to store persistent cookies; applications may use a flat text file, a database or any other storage method available.

In addition to the cookie name and value, the server may return additional information about the cookie which the client should use to determine if it should send the cookie back to the server:

The ***cookie path*** specifies a path for the resources where the cookie should be used. For example, a path of "/" indicates that the cookie should be provided for all resources requested from the server. A path of "/data" would mean that the cookie should be included if the resource is found in the /data folder or a sub-folder, such as /data/projections.asp. However, the cookie would not be provided if the resource /info/status.asp was requested, since it is not in the /data path.

The ***cookie domain*** specifies the domain for which the cookie should be used. Matches are made by comparing the name of the server against the domain name specified in the cookie. If the domain is example.com, then any server in the example.com domain would match; for example, both shipping.example.com and orders.example.com would match the domain value. However, if the cookie domain was orders.example.com, then the cookie would only be sent if the resource was requested from orders.example.com, not if the resource was located on shipping.example.com or www.example.com.

The ***cookie expiration*** specifies the date and time that the cookie should be deleted and no longer sent when a resource is requested from the server. This is only valid for persistent cookies, since session cookies are automatically deleted when the client application

terminates. The time is always expressed as Coordinated Universal Time.

The ***cookie flags*** provide additional information about the cookie. In some cases, a cookie should only be submitted to the server if the resource is requested using a secure connection. In this case, the bit flag HTTP_COOKIE_SECURE will be set.

It is the responsibility of the client application to determine if a cookie meets the criteria required to be submitted to the server. If the application wishes to send the cookie, it can use the **HttpSetCookie** function and specify the cookie name and value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpClearHeaders, HttpGetFirstCookie, HttpGetNextCookie, HttpGetResponseHeader, HttpSetCookie, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetData Function

```
INT WINAPI HttpGetData(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);
```

The **HttpGetData** function requests a resource from the server and copies the data to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to

	accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **HttpEnableCompression** function.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetData** function is used to retrieve a resource from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated and the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpdwLength*** parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the GET request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

When encountering a server error during a request, the **HttpGetData** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. Additionally, servers may provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_TRANSFER_ERRORDATA option. When this option is enabled, the behavior of **HttpGetData** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the buffer provided by the caller.

If compression has been enabled and the server returns compressed data, it will be automatically expanded before being returned to the caller. This will result in a difference between the value returned in the ***lpdwLength*** parameter, which contains the actual number of bytes copied into the buffer, and the values reported by **HttpGetTransferStatus**. For example, if the server returns 5,000 bytes of compressed data that expands into 15,000 bytes, this function will return 15,000 as the number of bytes copied into the buffer. However, the **HttpGetTransferStatus** function will return the

content length as the original 5,000 bytes of compressed data. For this reason, you should always use the value returned in the *lpdwLength* parameter to determine the amount of data that has been copied into the buffer.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the response in a block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = HttpGetData(hClient,
                      lpszResource,
                      &hgblBuffer,
                      &cbBuffer,
                      HTTP_TRANSFER_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableCompression, HttpEnableEvents, HttpGetFile, HttpGetTransferStatus, HttpPostData, HttpPutData, HttpPutFile, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetDefaultUserAgent Function

```
INT WINAPI HttpGetDefaultUserAgent(
    LPTSTR lpszUserAgent,
    INT nMaxLength
);
```

The **HttpGetDefaultUserAgent** function returns the default user agent string which is included with all requests.

Parameters

lpszUserAgent

A buffer which will contain the default user agent string when the function returns. The buffer must be large enough to contain the entire user agent string including the terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When you submit a request to the server, a header named `User-Agent` is automatically defined which identifies your application. It typically uses a format where you have a product name and version number separated by a slash. There can be multiple products listed in the user agent string, with additional optional information enclosed in parenthesis.

A server uses the user agent to determine what type of client has issued the request, including what operating system and browser version was used. The **HttpGetDefaultUserAgent** function returns a copy of that default value. The value of the user agent string depends on which version of Windows the client is running on and whether the process is 32-bit or 64-bit. For example, a 64-bit application will typically return a value like this:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/134.0.0.0
Safari/537.36 SocketTools/12.0
```

Some web services check the value of the user agent string to determine if a compatible client is being used to issue the request. The default value is designed to emulate a common browser, but some services may require you change the user agent to use a specific value or include certain product names and versions.

You can change the value of the user agent string in one of two ways. You can use **HttpSetRequestHeader** to set the `User-Agent` header to a specific value for a client session, or you can use **HttpSetDefaultUserAgent** to set the default user agent string for all requests. If you use **HttpSetRequestHeader** to change the user agent string for a client session, it will always override the default value.

To prevent any user agent string from being included with a request, include the HTTP_OPTION_NOUSERAGENT option when connecting to the server or use **HttpSetOption** to

enable that option after a connection has been established.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpSetDefaultUserAgent, HttpSetOption, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetDefaultZoneId Function

```
BOOL WINAPI HttpGetDefaultZoneId(
    HCLIENT hClient,
    LPDWORD lpdwZoneId
);
```

The **HttpGetDefaultZoneId** function returns the default security zone for downloaded files.

Parameters

hClient

Handle to the client session.

lpdwZoneId

A pointer to an unsigned integer which is updated with the default zone ID for the client session. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the function succeeds, the return value is non-zero and *lpdwZoneId* will be updated with the default zone identifier. If no default zone has been explicitly set, the function will still return success, and the variable *lpdwZoneId* points to will be set to `SECURITY_ZONE_NONE`. If the function fails, it returns zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetDefaultZoneId** function returns the zone identifier which was previously set using the **HttpSetDefaultZoneId** function. If no default zone has been specified for the client session, the default behavior is to not set a security zone for the file. This maintains backwards compatibility with previous versions of the library which would not create the security zone metadata for local files.

This function is typically used to confirm the currently configured zone before downloading a file, or to verify whether the default behavior has been overridden. The default zone ID is specific to the client session and does not affect other sessions or system-wide settings.

To get the security zone identifier for a local file, use the **HttpGetFileZoneInfo** function. To change or remove the security zone information for a file, use the **HttpSetFileZoneInfo** function.

For more detailed information about the zone identifiers, refer to the **HttpSetDefaultZoneId** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetFileZoneInfo, HttpSetDefaultZoneId, HttpSetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetEncodingType Function

```
INT WINAPI HttpGetEncodingType(
    HCLIENT hClient
);
```

The **HttpGetEncodingType** function determines which content-encoding option is enabled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is one of the following values:

Value	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request and no default content type will be specified. This encoding type should be used with REST APIs and other services which expect XML or JSON request payloads.
HTTP_ENCODING_URL	Non-printable and extended ASCII characters will be encoded so they can be safely used with URLs and form data. Encoded characters will be represented by a percent symbol prefix, followed by a two digit hexadecimal value which represents the ASCII character code. This encoding is typically used with web services which process HTML form data.
HTTP_ENCODING_XML	This encoding is identical to URL encoding, except spaces are not encoded. It is used with legacy web services which expect form data in an XML format and cannot process encoded whitespace. This encoding should not be specified for services which use REST APIs.

If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The default encoding type is HTTP_ENCODING_URL which is used for submitting data to services which expect HTML form data using the **HttpPostData** or **HttpSubmitForm** functions.

When submitting a JSON or XML request to a service using a REST API, your application should use HTTP_ENCODING_NONE and set the appropriate content type for the request payload. The HTTP_ENCODING_XML encoding type should only be used if the server expects URL encoded form data. The **HttpPostJson** and **HttpPostXml** functions will automatically set the correct encoding and content type for those requests.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetContentType, HttpPostData, HttpPostJson, HttpPostXml, HttpSetEncodingType, HttpSubmitForm

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetErrorString Function

```
INT WINAPI HttpGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **HttpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **HttpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetLastError, HttpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFile Function

```
INT WINAPI HttpGetFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   DWORD dwOffset
);
```

The **HttpGetFile** function transfers the specified file on the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local system.

lpszRemoteFile

A null-terminated string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **HttpEnableCompression** function. This option is ignored if the ***dwOffset*** parameter is non-zero.

HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

dwOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to download the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. To determine if the server compressed the data returned to the client, use the **HttpGetResponseHeader** function to get the value of the **Content-Encoding** header after this function returns. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

To download large files that are over 4GB, use the **HttpGetFileEx** function.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
LPCTSTR lpszLocalFile = _T("index.html");
LPCTSTR lpszRemoteFile = _T("http://sockettools.com/");

if (HttpGetFile(hClient, lpszLocalFile, lpszRemoteFile) == HTTP_ERROR)
{
    dwError = HttpGetLastError();
    _tprintf(stderr, "Unable to download %s, error 0x%lx\n", lpszRemoteFile,
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetData, HttpGetFileEx, HttpGetText, HttpGetTransferStatus, HttpPutFile, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFileEx Function

```
INT WINAPI HttpGetFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions,
   ULARGE_INTEGER uiOffset
);
```

The **HttpGetFileEx** function transfers the specified file on the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the **HttpEnableCompression** function. This option is ignored if the ***uiOffset*** parameter is

	non-zero.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to download the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. To determine if the server compressed the data returned to the client, use the **HttpGetResponseHeader** function to get the value of the **Content-Encoding** header after this function returns. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetData, HttpGetText, HttpGetTransferStatusEx, HttpPutFileEx, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFilePathUrl Function

```
INT WINAPI HttpGetFilePathUrl(
   HCLIENT hClient,
   LPCTSTR lpszFilePath,
   LPTSTR lpszFileUrl,
   INT nMaxLength
);
```

The **HttpGetFilePathUrl** function returns a normalized URL for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFilePath

A null-terminated string which specifies the file path on the server. The path may be absolute or relative, and the path does not have to exist on the server. Although this may be any type of resource, this function is primarily intended for use with static content such as documents and images. This parameter cannot be NULL.

lpszFileUrl

A null-terminated string buffer that will contain the URL for the specified file. If this parameter is NULL, nothing will be copied but the function will return the length of the URL, not including the terminating null character.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. This must include space for the terminating null character. If ***lpszFileUrl*** is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the ***lpszFileUrl*** buffer, not including the terminating null character. If a buffer is provided, it must be large enough to contain the entire URL or the function will fail. If the ***lpszFileUrl*** parameter is NULL, the function will return the length of the URL and nothing will be copied. If the function fails, it will return zero. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpGetFilePathUrl** function will return the URL associated with the specified file path. This can be useful if you wish to display or store URLs rather than relative paths to a file on the current server. For example, if you are connected to the server `example.com` on port 443 using a secure connection and you provide a file path such as `/documents/invoice.pdf` this function would return a URL which looks like `https://example.com/documents/invoice.pdf`. For security reasons, the resulting URL will never contain user credentials.

The file path in the URL will be normalized to remove extraneous directory traversals and path separators. It is not required for the file path to exist on the server at the time this function is called. For more information about path normalization, refer to the **HttpNormalizePath** function.

This function can also be used to obtain a URL which can be passed to the **HttpGetSercurityZoneId** function to determine which security zone should be assigned to the file when it is downloaded.

Example

The following example shows how **HttpGetFilePathUrl** could be used to obtain the origin URL for a file on the server and then use that in conjunction with the **HttpGetSecurityZoneId** function to update the security zone metadata for the file if the download is successful.

```
LPCTSTR lpszLocalFile = _T("\\Users\\Public\\Documents\\Document.pdf");
LPCTSTR lpszRemoteFile = _T("/files/document.pdf");

TCHAR szFileUrl[FTP_MAXURLPATHLEN] = { 0 };
DWORD dwZoneId = SECURITY_ZONE_NONE;

// Return a complete URL for the relative file path specified for the
// remote file on the server
if (HttpGetFilePathUrl(hClient, lpszRemoteFile, szFileUrl, FTP_MAXURLPATHLEN) < 1)
{
    _tprintf(_T("Unable to get URL for \"%s\"\n"), lpszRemoteFile);
    return;
}

// Check the security zone for the remote file URL
if (!HttpGetSecurityZoneId(szFileUrl, &dwZoneId))
{
    _tprintf(_T("Unable to determine a security zone for \"%s\"\n"), szFileUrl);
}

// Download the file from the server
INT nResult = HttpGetFile(hClient, lpszLocalFile, lpszRemoteFile,
HTTP_TRANSFER_DEFAULT, 0);

if (nResult != FTP_ERROR)
{
    // The download was successful; if a security zone was returned for
    // the file, then update it with the zone ID and origin URL
    if (dwZoneId != SECURITY_ZONE_NONE)
    {
        if (!HttpSetFileZoneInfo(lpszLocalFile, dwZoneId, szFileUrl))
        {
            _tprintf(_T("Unable to set the security zone for \"%s\"\n"),
lpszLocalFile);
            return;
        }
    }
}
else
{
    // The download failed
    _tprintf(_T("Unable to download \"%s\" to \"%s\"\n"), lpszRemoteFile,
lpszLocalFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpGetSecurityZoneId, HttpNormalizePath, HttpNormalizeUrl, HttpValidateUrl

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFileSize Function

```
INT WINAPI HttpGetFileSize(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPDWORD lpdwFileSize
);
```

The **HttpGetFileSize** function returns the size of the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. If the file does not exist or the client does not have permission to access the file, this method will fail and the ***lpdwFileSize*** parameter will be set to zero.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the function will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the ***lpdwFileSize*** parameter may still contain a valid value. If the library was able to determine the file size, but was not able to maintain the connection to the server, the returned file size will be greater than zero even if the function returns an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCommand, HttpGetContentLength, HttpGetContentType, HttpGetFileTime, HttpVerifyFile

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFileSizeEx Function

```
INT WINAPI HttpGetFileSizeEx(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   ULARGE_INTEGER *lpFileSize
);
```

The **HttpGetFileSizeEx** function returns the size of the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileSize

Points to an unsigned 64-bit integer that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. If the file does not exist or the client does not have permission to access the file, this method will fail and the ***lpFileSize*** parameter will be set to zero.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the function will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the ***lpFileSize*** parameter may still contain a valid value. If the library was able to determine the file size, but was not able to maintain the connection to the server, the returned file size will be greater than zero even if the function returns an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCommand, HttpGetFileTime, HttpVerifyFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFileTime Function

```
INT WINAPI HttpGetFileTime(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPSYSTEMTIME lpFileTime,
   BOOL bLocalize
);
```

The **HttpGetFileTime** function returns the modification time for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a SYSTEMTIME structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetFileTime** function can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time.

This function uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a modification time for some resources. If the request to the server was successful and the file exists, but the server does not return a modification time, the function will succeed but all of the members of the SYSTEMTIME structure will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the modification time and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the SYSTEMTIME structure may still contain a valid value. If the library was able to determine the modification time, but was not able to maintain the connection to the server, the members of the SYSTEMTIME structure will specify a valid date and time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCommand, HttpGetFileSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFileZoneInfo Function

```
BOOL WINAPI HttpGetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   LPDWORD lpdwZoneId,
   LPTSTR lpszHostUrl,
   INT nMaxLength
);
```

The **HttpGetFileZoneInfo** function returns the security zone information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and read from the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

lpdwZoneId

A pointer to an unsigned integer which will contain the zone ID assigned to the file when the function returns. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string buffer which will contain the URL used to originally download the file. This information is not always available, in which case an empty string will be returned. If this information is not required, this parameter may be NULL and the ***nMaxLength*** parameter should be specified with a value of zero.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the ***lpszHostUrl*** string including the terminating null character. If the ***lpszHostUrl*** parameter is NULL, this value should be zero. If the buffer is not large enough to contain the entire URL, the string will be truncated. The maximum allowed URL length is 2,048 characters.

Return Value

If the function succeeds, the return value is non-zero. If the function fails or zone information is not available, the function returns zero. To get extended error information, call **HttpGetLastError**.

Remarks

When a file is downloaded from the Internet or other external sources, Windows may assign a zone identifier to the file using an alternate data stream named `Zone.Identifier`. This metadata helps determine how the system treats the file with respect to security, such as whether to prompt the user with a warning before opening it.

The **HttpGetFileZoneInfo** function retrieves the zone ID associated with the specified local file, allowing an application to determine the origin and trust level of that file. This can be useful when implementing additional security measures, auditing downloaded content, or providing user warnings about potentially untrusted files.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be read from files on network volumes because the SMB protocol does not support alternate data streams.

In addition to the zone ID, this function can optionally return the source URL used to download the file. This information, if present, is extracted from the same alternate data stream. Not all files will include this URL, as it depends on the program or browser that downloaded the file and whether it preserved this metadata. This function will not return an error if the URL stored with the security zone information is not a valid HTTP URL. For example, it's possible for this function to return zone information with a FTP URL. You can use the **HttpValidateUrl** function to determine if the host URL is a valid HTTP URL.

Note that not all file systems support alternate data streams. For example, FAT32 volumes do not support them, and in such cases, zone information will not be available. Additionally, some applications may remove this metadata after download or strip it during file copy operations. The absence of zone information does not guarantee that a file is safe or originated locally.

If you wish to distinguish between a file which exists without any zone information, and an error condition where the file does not exist or could not be accessed, check the last error code. If this function returns zero and the **HttpGetLastError** function returns NO_ERROR, this indicates the file name is valid and it exists, but the `Zone.Identifier` alternate data stream does not exist.

For more detailed information about the security zone identifiers, refer to the **HttpSetDefaultZoneId** function. To change or remove the security zone metadata from a local file, use the **HttpSetFileZoneInfo** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDownloadFile, HttpGetDefaultZoneId, HttpSetDefaultZoneId, HttpSetFileZoneInfo, HttpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFirstCookie Function

```
DWORD_PTR WINAPI HttpGetFirstCookie(
    HCLIENT hClient,
    LPTSTR lpszCookieName,
    LPINT lpnNameLen,
    LPTSTR lpszCookieValue,
    LPINT lpnValueLen
);
```

The **HttpGetFirstCookie** function returns the first cookie set by the server.

Parameters

hClient

Handle to the client session.

lpszCookieName

A null-terminated string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A null-terminated string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the function succeeds, the return value is an opaque token which will be passed to the **HttpGetNextCookie** function to retrieve the next cookie. If there are no cookies or the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetFirstCookie** function is used to enumerate the cookies set by the server after a resource has been requested. The return value is an unsigned integer where the size depends on whether the application targets 32-bit or 64-bit platforms. This value should always be treated as an opaque integer which is only valid during the enumeration of the session cookies and should never be stored by the application.

To get information about a specific cookie, use the **HttpGetCookie** function.

Example

```
TCHAR szCookieName[MAXCOOKIENAME];
TCHAR szCookieValue[MAXCOOKIEVALUE];
INT nNameLen, nValueLen;
DWORD_PTR dwCookie;
```

```
// Initialize the nNameLen and nValueLen variables to
// the maximum number of characters that can be copied
// into the string buffers
nNameLen = MAXCOOKIENAME;
nValueLen = MAXCOOKIEVALUE;

// Get the first cookie set by the server
dwCookie = HttpGetFirstCookie(hClient,
                              szCookieName,
                              &nNameLen,
                              szCookieValue,
                              &nValueLen);

while (dwCookie != 0)
{
    // The szCookieName and szCookieValue strings contain the
    // the name and value for a cookie set by the server

    // Re-initialize the nNameLen and nValueLen variables
    // to the maximum length of the strings
    nNameLen = MAXCOOKIENAME;
    nValueLen = MAXCOOKIEVALUE;

    // Get the next cookie set by the server
    dwCookie = HttpGetNextCookie(hClient,
                                 dwCookie,
                                 szCookieName,
                                 &nNameLen,
                                 szCookieValue,
                                 &nValueLen);
};
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCookie, HttpGetNextCookie, HttpGetResponseHeader, HttpSetCookie, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFirstHeader Function

```
BOOL WINAPI HttpGetFirstHeader(
    HCLIENT hClient,
    UINT nHeaderType,
    LPTSTR lpszHeader,
    LPINT lpcchHeader
    LPTSTR lpszValue,
    LPINT lpcchValue
);
```

The **HttpGetFirstHeader** function returns the name and value of the first request or response header.

Parameters

hClient

Handle to the client session.

nHeaderType

Specifies the type of header to return. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Return the first header set by the client before a resource has been requested. Request headers typically specify information that the server needs to complete the request, such as the name of the host that has the resource, or what types of resources are acceptable to the client.
HTTP_HEADERS_RESPONSE	Return the first header set by the server after a request has been submitted to the server. The response headers typically contain information about the resource, such as its size, the content type and the date it was last modified.

lpszHeader

A null-terminated string buffer that will contain the name of the header field when the function returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header name string.

lpszValue

A null-terminated string buffer that will contain the name of the header value when the function returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header value string.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call

HttpGetLastError.

Remarks

Use this function together with **HttpGetNextHeader** to enumerate all request or response headers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpSetRequestHeader, HttpGetNextHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFirstQueuedFile Function

```
DWORD WINAPI HttpGetFirstQueuedFile(
   HQUEUE hQueue,
   DWORD dwQueueMode,
   DWORD dwExcludeFiles,
   LPHTTPQUEUEDFILE lpFileInfo
);
```

The **HttpGetFirstQueuedFile** function returns information about the first file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An unsigned integer value which specifies if the file name should be matched against downloaded files, uploaded files or all files in the queue. It may be one of the following values:

Constant	Description
HTTP_QUEUE_ALL	Return information about all files in the queue.
HTTP_QUEUE_DOWNLOAD	Return information for files which are queued for download.
HTTP_QUEUE_UPLOAD	Return information for files which are queued for upload.

dwExcludeFiles

An unsigned integer value which specifies which files to exclude from the search. If this value is zero, no files will be excluded from the search; otherwise, this value is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_QUEUE_FLAG_NONE	No files should be excluded.
HTTP_QUEUE_FLAG_COMPLETED	Exclude files which have been transferred. This flag is set after a queued file has been uploaded or downloaded successfully.
HTTP_QUEUE_FLAG_FAILED	Exclude files which where not successfully uploaded or downloaded. This flag is set when an error occurs, either when establishing a connection with the server or during the file transfer.
HTTP_QUEUE_FLAG_CANCELED	Exclude files which were canceled during the file transfer. This flag is only set when the **HttpCancelQueuedFile** function has been called and a queued file is in the process of being uploaded or downloaded.

lpFileInfo

A pointer to a HTTPQUEUEDFILE structure which will contain information about the file. This

parameter cannot be NULL.

Return Value

If the function succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no queued files which match the specified parameters, or the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns information about the first file in the current transfer queue. It is used in conjunction with the **HttpGetNextQueuedFile** function to obtain information about all queued files.

Example

```
// List all files in a file transfer queue
HTTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = HttpGetFirstQueuedFile(hQueue, HTTP_QUEUE_ALL, HTTP_QUEUE_FLAG_NONE,
&queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case HTTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case HTTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
    }

    dwFileId = HttpGetNextQueuedFile(hQueue, &queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpEnumQueuedFiles, HttpFindQueuedFile, HttpGetNextQueuedFile, HttpGetQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetFormProperties Function

```
INT WINAPI HttpGetFormProperties(
   HFORM hForm,
   LPHTTPFORMPROPERTIES lpFormProp
);
```

The **HttpGetFormProperties** function returns information about the specified form.

Parameters

hForm

Handle to the virtual form.

lpFormProp

Points to a HTTPFORMPROPERTIES structure which will contain information about the specified form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCreateForm, HttpSetFormProperties, HttpSubmitForm, HTTPFORMPROPERTIES

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetLastError Function

```
DWORD WINAPI HttpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **HttpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **HttpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **HttpSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those functions which call **HttpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetErrorString, HttpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetNextCookie Function

```
DWORD_PTR WINAPI HttpGetNextCookie(
   HCLIENT hClient,
   DWORD_PTR dwCookie,
   LPTSTR lpszCookieName,
   LPINT lpnNameLen,
   LPTSTR lpszCookieValue,
   LPINT lpnValueLen
);
```

The **HttpGetNextCookie** function returns the next cookie set by the server.

Parameters

hClient

Handle to the client session.

dwCookie

An unsigned integer value which specifies a cookie token returned by a previous call to either **HttpGetFirstCookie** or **HttpGetNextCookie**.

lpszCookieName

A null-terminated string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A null-terminated string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie value string. If the ***lpszCookieValue*** parameter is NULL, this parameter should also be NULL.

Return Value

If the function succeeds, the return value is an opaque token which will be passed to the **HttpGetNextCookie** function to retrieve the next cookie. If there are no more cookies or the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetNextCookie** function is used to enumerate the cookies set by the server after a resource has been requested. The return value is an unsigned integer where the size depends on whether the application targets 32-bit or 64-bit platforms. This value should always be treated as an opaque integer which is only valid during the enumeration of the session cookies and should never be stored by the application.

To get information about a specific cookie, use the **HttpGetCookie** function.

Example

```
TCHAR szCookieName[MAXCOOKIENAME];
TCHAR szCookieValue[MAXCOOKIEVALUE];
INT nNameLen, nValueLen;
DWORD_PTR dwCookie;

// Initialize the nNameLen and nValueLen variables to
// the maximum number of characters that can be copied
// into the string buffers
nNameLen = MAXCOOKIENAME;
nValueLen = MAXCOOKIEVALUE;

// Get the first cookie set by the server
dwCookie = HttpGetFirstCookie(hClient,
                              szCookieName,
                              &nNameLen,
                              szCookieValue,
                              &nValueLen);

while (dwCookie != 0)
{
    // The szCookieName and szCookieValue strings contain the
    // the name and value for a cookie set by the server

    // Re-initialize the nNameLen and nValueLen variables
    // to the maximum length of the strings
    nNameLen = MAXCOOKIENAME;
    nValueLen = MAXCOOKIEVALUE;

    // Get the next cookie set by the server
    dwCookie = HttpGetNextCookie(hClient,
                                 dwCookie,
                                 szCookieName,
                                 &nNameLen,
                                 szCookieValue,
                                 &nValueLen);
};
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCookie, HttpGetFirstCookie, HttpGetResponseHeader, HttpSetCookie, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetNextHeader Function

```
BOOL WINAPI HttpGetNextHeader(
    HCLIENT hClient,
    UINT nHeaderType,
    LPTSTR lpszHeader,
    LPINT lpcchHeader,
    LPTSTR lpszValue,
    LPINT lpcchValue
);
```

The **HttpGetNextHeader** function returns the name and value of the next request or response header.

Parameters

hClient

Handle to the client session.

nHeaderType

Specifies the type of header to return. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Return the first header set by the client before a resource has been requested. Request headers typically specify information that the server needs to complete the request, such as the name of the host that has the resource, or what types of resources are acceptable to the client.
HTTP_HEADERS_RESPONSE	Return the first header set by the server after a request has been submitted to the server. The response headers typically contain information about the resource, such as its size, the content type and the date it was last modified.

lpszHeader

A null-terminated string buffer that will contain the name of the header field when the function returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header name string.

lpszValue

A null-terminated string buffer that will contain the name of the header value when the function returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header value string.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client

handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

Use this function iteratively after **HttpGetFirstHeader** to enumerate all request or response headers.

Unlike the **HttpGetResponseHeader** function, which returns a single header name and value, the **HttpGetFirstHeader** and **HttpGetNextHeader** functions will return multiple headers that have the same common name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpSetRequestHeader, HttpGetFirstHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetNextQueuedFile Function

```
DWORD WINAPI HttpGetNextQueuedFile(
   HQUEUE hQueue,
   LPHTTPQUEUEDFILE lpFileInfo
);
```

The **HttpGetNextQueuedFile** function returns information about the next file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

lpFileInfo

A pointer to a HTTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is an unsigned integer with a non-zero value which uniquely identifies the file in the queue. If there are no additional queued files which match the exclusion criteria specified with the call to **HttpGetFirstQueuedFile**, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns information about the next file in the current transfer queue after an initial call to **HttpGetFirstQueuedFile**. To obtain a list of all matching files in the queue, call this function repeatedly until it returns a value of zero. When information about the last file in the queue has been returned, the last error code will be set to `ST_ERROR_END_OF_QUEUE`.

Example

```
// List all files in a file transfer queue
HTTPQUEUEDFILE queuedFile;
DWORD dwFileId;

dwFileId = HttpGetFirstQueuedFile(hQueue, HTTP_QUEUE_ALL, HTTP_QUEUE_FLAG_NONE,
&queuedFile);

while (dwFileId != 0)
{
    switch (queuedFile.dwQueueMode)
    {
    case HTTP_QUEUE_DOWNLOAD:
        _tprintf(_T("%u: download \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;

    case HTTP_QUEUE_UPLOAD:
        _tprintf(_T("%u: upload \"%s\" to \"%s\"\n"),
                 queuedFile.dwFileId,
                 queuedFile.szRemoteFile,
                 queuedFile.szLocalFile);
        break;
```

```
    }

    dwFileId = HttpGetNextQueuedFile(hQueue, &queuedFile);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpEnumQueuedFiles, HttpFindQueuedFile, HttpGetFirstQueuedFile, HttpGetQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetOption Function

```
INT WINAPI HttpGetOption(
   HCLIENT hClient,
   DWORD dwOption,
   LPBOOL lpbEnabled
);
```

The **HttpGetOption** function determines whether a specified HTTP option is enabled.

Parameters

hClient

Handle to the client session.

dwOption

An unsigned integer which specifies the option that is to be checked. It may be one of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the

	connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpbEnabled

A pointer to an integer which will be set to a non-zero value when the function returns if the specified option has been enabled. If the option has not been enabled, a zero value will be returned in the variable.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**. Note that the value returned in the

lpbEnabled parameter is only valid if the function succeeds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpConnect, HttpSetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetPriority Function

```
INT WINAPI HttpGetPriority(
   HCLIENT hClient
);
```

The **HttpGetPriority** function returns a value which specifies the priority of file transfers.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current file transfer priority. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetPriority** function can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Value	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpSetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetQueueClient Function

```
HCLIENT WINAPI HttpGetQueueClient(
     HQUEUE hQueue
);
```

The **HttpGetQueueClient** function returns a handle to the current client session.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value a handle to a client session. If the function fails, it will return INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetQueueClient** function returns the handle to a client session which may be used with other functions, such as **HttpGetTransferStatus**. A valid handle will only be returned if the queue manager is currently uploading or downloading a file. If the queue is active but there are no file transfers in progress at the time this function is called, it will fail and set the last error code to `ST_ERROR_NO_QUEUED_TRANSFER`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetTransferStatus, HttpGetTransferStatusEx, HttpGetQueueStatus, HttpGetThreadQueue, HttpStartQueue, HttpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetQueuedFile Function

```
BOOL WINAPI HttpGetQueuedFile(
   HQUEUE hQueue,
   DWORD dwFileId,
   LPHTTPQUEUEDFILE lpFileInfo
);
```

The **HttpGetQueuedFile** function returns information about the specified file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

lpFileInfo

A pointer to a HTTPQUEUEDFILE structure which will contain information about the file. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the file identifier is not valid or the file has been removed from the queue, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The file identifier can be obtained by several different functions, including **HttpEnumQueuedFiles** and **HttpGetQueueStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

Example

```
// Get the number of files in the queue
LONG nFiles = HttpEnumQueuedFiles(hQueue, NULL, 0)

// Display information about the queued files
if (nFiles > 0)
{
    DWORD *pdwFileList = new DWORD[nFiles];

    nFiles = HttpEnumQueuedFiles(hQueue, pdwFileList, nFiles);

    for (LONG nIndex = 0; nIndex < nFiles; nIndex++)
    {
        HTTPQUEUEDFILE queuedFile;

        if (HttpGetQueuedFile(hQueue, pdwFileList[nIndex], &queuedFile) == FALSE)
            break;

        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
```

```
                _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                 break;

        case HTTP_QUEUE_UPLOAD:
                _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        queuedFile.dwFileId,
                        queuedFile.szLocalFile,
                        queuedFile.szRemoteFile);
                 break;
        }
    }

    delete pdwFileList;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpEnumQueuedFiles, HttpFindQueuedFile, HttpGetFirstQueuedFile, HttpGetNextQueuedFile, HttpGetQueueStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetQueueStatus Function

```
INT WINAPI HttpGetQueueStatus(
   HQUEUE hQueue,
   LPHTTPQUEUESTATUS lpQueueStatus
);
```

The **HttpGetQueuedFile** function returns information about the specified file in the transfer queue.

Parameters

hQueue

A handle to a file transfer queue.

lpQueueStatus

A pointer to a HTTPQUEUESTATUS structure which will contain information about the current state of the file transfer queue. If this parameter is NULL, the function will ignore the parameter and only return the current status of the queue.

Return Value

If the function succeeds, the return value is the current queue status. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**. The following status values may be returned:

Value	Description
HTTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before **HttpStartQueue** has been called or after **HttpStopQueue** has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
HTTP_QUEUE_STATUS_ACTIVE	The queue manager is active and files are currently being uploaded or downloaded.
HTTP_QUEUE_STATUS_PAUSED	The queue manager is active although file transfers are currently paused. The queue enters this state after the **HttpSuspendQueue** function is called and resumes file transfers after the **HttpResumeQueue** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCancelQueuedFile, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpSuspendQueue, HttpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetQueueThread Function

```
DWORD WINAPI HttpGetQueueThread(
    HQUEUE hQueue
);
```

The **HttpGetQueueThread** function returns a value which uniquely identifies the queue manager thread.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero and specifies a unique identifier for the queue manager thread. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetQueueThread** function can be used to obtain a unique identifier for the queue manager thread which is responsible for performing the file transfers. If the function returns zero, this indicates that either the queue handle is no longer valid, or the queue is idle and no files are currently being processed. To obtain the handle to a queue associated with a particular thread, call the **HttpGetThreadQueue** function.

You should not store the thread ID or consider it a persistent value, even during the lifetime of the process. It is only valid while the queue manager is actively processing files in the queue. The thread ID value will change every time the **HttpStartQueue** function is called.

Never use the thread ID to obtain a handle to the thread and call Windows API functions such as **SuspendThread** or **TerminateThread**. This can result inconsistency within the internal state of the queue manager and may result in incomplete or corrupted file transfers.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetQueueStatus, HttpGetThreadQueue, HttpStartQueue, HttpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetRedirectUrl Function

```
INT WINAPI HttpGetRedirectUrl(
   HCLIENT hClient,
   LPTSTR lpszResource,
   INT nMaxLength
);
```

The **HttpGetRedirectUrl** function returns the URL for a redirected resource.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string buffer which will contain the URL of the redirected resource when the function returns. The string will be null-terminated and the buffer must be large enough to accommodate the entire URL or the function will fail. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the length of the URL. A return value of zero indicates that the request was not redirected. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns the complete URL for a request which has been redirected to a different resource. This is useful for applications which do not wish to use automatic redirection and instead prefer to handle the redirection by establishing a new connection by passing this value to the **HttpConnectUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpGetResourceUrl, HttpGetResponseHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetRequestHeader Function

```
BOOL WINAPI HttpGetRequestHeader(
    HCLIENT hClient,
    LPCTSTR lpszHeader,
    LPTSTR lpszValue,
    INT nMaxLength
);
```

The **HttpGetRequestHeader** function returns the value of the specified request header field.

Parameters

hClient

Handle to the client session.

lpszHeader

A pointer to a null-terminated string which specifies the name of the header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszValue

Pointer to a buffer which will contain the null-terminated string value of the specified header field.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

When a resource is requested from the server, it consists of two parts. The first part is the command issued to the server, along with the name of the resource and any optional encoded parameters. The second part consists of one or more header fields which can be used to provide additional information to the server. For example, here is what the command and header may look like for a simple request:

```
GET /test HTTP/1.0
Host: api.sockettools.com
Accept: text/*
```

The first line consists of the command, the resource and the protocol version. The subsequent lines are the header, which is similar to the headers used in email messages. The Host field specifies the name of the server the resource is being requested from, and the Accept field specifies the type of resources that are acceptable to the client; in this case, any type of text file.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetResourceUrl Function

```
INT WINAPI HttpGetResourceUrl(
   HCLIENT hClient,
   LPTSTR lpszResource,
   INT nMaxLength
);
```

The **HttpGetResourceUrl** function returns the URL for the current request.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string buffer which will contain the URL of the current resource when the function returns. The string will be null terminated and the buffer must be large enough to accommodate the entire URL or the function will fail. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the length of the URL. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns the URL for the current request which has been submitted to the server. This can be useful for applications which need to compose a URL for display or logging purposes, combining the connection information and resource path provided to functions such as **HttpGetData** or **HttpPostData**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpGetData, HttpGetRedirectUrl, HttpPostData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetResponseHeader Function

```
BOOL WINAPI HttpGetResponseHeader(
    HCLIENT hClient,
    LPCTSTR lpszHeader,
    LPTSTR lpszValue,
    INT nMaxLength
);
```

The **HttpGetResponseHeader** function returns the value of the specified response header field.

Parameters

hClient

Handle to the client session.

lpszHeader

A pointer to a null-terminated string which specifies the name of the header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszValue

Pointer to a buffer which will contain the null-terminated string value of the specified header field.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

When a resource is returned by the server, it consists of three parts. The first part consists of a single line that indicates the result of the request. The second part is one or more header fields which provides specific information about the resource, such as its size in bytes. The third part consists of the resource data itself, such as the HTML document or image data. For example, this is what the response to a request for a simple HTML document can look like:

```
HTTP/1.0 200 OK
Date: Mon, 5 Jan 2004 20:18:33 GMT
Content-Type: text/html
Last-Modified: Mon, 5 Jan 2004 19:34:19 GMT
Content-Length: 115

<html>
<head>
<title>Simple Document</title>
</head>
<body>
This is a simple HTML document.
<body>
</html>
```

The first line consists of the protocol version, a numeric response code and some text describing the

result. The subsequent lines are the header, which is similar to the headers used in email messages. For example, the Date field specifies the date the resource was requested, the Content-Type field specifies what type of resource was requested, and the Content-Length field specifies the size of the resource in bytes. The end of the header block is indicated by an empty line (two carriage-return/linefeed sequences), and is followed by the resource itself, in this case a simple HTML document.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetContentLength, HttpGetContentType, HttpGetCookie, HttpGetFirstCookie, HttpGetFirstHeader, HttpGetNextCookie, HttpGetNextHeader, HttpGetRequestHeader, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetResultCode Function

```
INT WINAPI HttpGetResultCode(
    HCLIENT hClient
);
```

The **HttpGetResultCode** function reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCommand, HttpGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetResultString Function

```
INT WINAPI HttpGetResultString(
    HCLIENT hClient,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

The **HttpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

The buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **HttpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCommand, HttpGetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetSecurityInformation Function

```
BOOL WINAPI HttpGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **HttpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (HttpGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
```

```
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnect, HttpDisconnect, HttpGetOption, HttpProxyConnect, HttpSetOption, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetSecurityZoneId Function

```
BOOL WINAPI HttpGetSecurityZoneId(
    LPCTSTR lpszResource,
    LPDWORD lpdwZoneId
);
```

The **HttpGetSecurityZoneId** function returns the security zone identifier for the specified resource.

Parameters

lpszResource

A null-terminated string which specifies the resource URL to check. This must be either a valid HTTP URL or a local file URL.

lpdwZoneId

A pointer to an unsigned integer which is updated with the zone ID for the specified resource. It may be one of the following values when the function returns:

Value	Description
SECURITY_ZONE_NONE	The security zone for this resource cannot be determined.
SECURITY_ZONE_LOCAL	The resource is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The resource originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The resource originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The resource originates from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The resource originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

Return Value

If the function succeeds, the return value is non-zero and ***lpdwZoneId*** will be updated with the zone identifier. If the security zone could not be determined, ***lpdwZoneId*** will be set to `SECURITY_ZONE_NONE`. If the function fails, it returns zero and the error code can be retrieved by calling **HttpGetLastError**.

Remarks

This **HttpGetSecurityZoneId** function is used to determine how a resource should be handled in terms of local security policy before performing a file transfer or download. For example, you may want to determine whether a resource originates from the Internet or local intranet before deciding whether to attach zone metadata, prompt the user, or apply additional security restrictions. It can also be used to implement policy-based rules for accepting or rejecting downloads based on their origin.

The ***lpszResource*** parameter must use the `http://`, `https://` or `file://` schemes. This function will not accept URLs for other schemes such as `ftp://` or `sftp://`. While this function does not verify that the resource exists, it does require the URL to be syntatically valid. If the host name cannot be resolved to an IPv4 or IPv6 address, the function will fail. If the resource is a local file, it must be a

regular file and cannot be the name of a folder or device.

If the resource is specified using a `file://` URL that refers to a file on a network share (e.g.: `file://server/share/file.txt`), the function will attempt to access that file across the network. This may cause the application to block while Windows resolves the hostname and establishes a connection to the shared volume. If the network share is unavailable or slow to respond, the function may block until the connection succeeds or the attempt times out.

If this function returns FALSE, this indicates either the resource type is not supported or there was an internal failure when the Internet Security Manager attempted to map the resource URL to a security zone. In most default configurations, this function will return the zone `SECURITY_ZONE_INTERNET` for a valid URL unless the host has been explicitly added to the **Intranet Zone** or **Trusted Sites** list on the local system.

Security zones are used with Mark of the Web (MOTW) metadata which indicates if a file originated from an external source, such as the Internet or an untrusted network. When a file is downloaded using a web browser or other applications, it may add a `Zone.Identifier` alternate data stream to the file. This stream contains information such as `ZoneId=3`, which corresponds to the Internet Zone. Your application can use the **HttpGetFileZoneInfo** function to retrieve this information, which allows you to determine whether a file was downloaded from an external source and what zone Windows has associated with it. This metadata affects how files are handled by Windows security components such as Microsoft Defender, SmartScreen, Office Protected View, and script execution policies in Windows Explorer. If the file has no metadata, it is generally treated as coming from the Local Machine Zone (zone 0). If your application sets or clears this metadata using the **HttpSetFileZoneInfo** function, it can influence how other applications interpret the file's trust level.

It is important to note that this function is used to determine what security zone would be appropriate for a given resource, and it only performs syntactic checks and classification of the URL using the **IInternetSecurityManager** COM interface. The security zone this function returns is based on the current system and user policy, using zone assignments stored in the registry. Typically this information is managed using Windows settings or Internet options in the control panel. This function does not read the zone identifier metadata which has been assigned to a file.

For more detailed information about the zone identifiers, refer to the **HttpSetDefaultZoneId** function.

Example

```
LPCTSTR pszDownloadUrl = _T("https://example.com/pub/download/file.txt");
DWORD dwZoneId = SECURITY_ZONE_NONE;

if (HttpGetSecurityZoneId(pszDownloadUrl, &dwZoneId))
{
    // dwZoneId now contains SECURITY_ZONE_INTERNET
    _tprintf(_T("The security zone is: %u\n"), dwZoneId);
}
else
{
    DWORD dwError = HttpGetLastError();
    // Handle the error
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetDefaultZoneId, HttpGetFileZoneInfo, HttpSetDefaultZoneId, HttpSetFileZoneInfo, HttpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetStatus Function

```
INT WINAPI HttpGetStatus(
   HCLIENT hClient
);
```

The **HttpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
HTTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
HTTP_STATUS_CONNECT	The client is establishing a connection with the server.
HTTP_STATUS_READ	The client is reading data from the server.
HTTP_STATUS_WRITE	The client is writing data to the server.
HTTP_STATUS_DISCONNECT	The client is disconnecting from the server.
HTTP_STATUS_GETDATA	The client is downloading data from the server.
HTTP_STATUS_PUTDATA	The client is uploading data to the server.
HTTP_STATUS_POSTDATA	The client is posting data to a script on the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session. To obtain status information about a file transfer, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpIsBlocking, HttpIsReadable, HttpIsWritable, HttpGetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetTaskError Function

```
DWORD WINAPI HttpGetTaskError(
    UINT nTaskId
);
```

Return the last error code for the specified asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed successfully, this function returns a value of zero. A non-zero return value indicates an error has occurred.

Remarks

The **HttpGetTaskError** function returns the last error code associated with the specified asynchronous task. If the task completed successfully, the return value will be zero. If the task is still active, the function will return the error ST_ERROR_TASK_ACTIVE. If the task has been suspended, the function will return ST_ERROR_TASK_SUSPENDED. Any other value indicates that the task completed, but the operation has failed and the error code will specify the cause of the failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskAbort, HttpTaskResume, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetTaskId Function

```
UINT WINAPI HttpGetTaskId(
   HCLIENT hClient
);
```

Return the asynchronous task identifier associated with the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the client handle is not associated with an asynchronous task, the function will return a value of zero.

Remarks

The **HttpGetTaskId** function will return the task ID that is associated with a client session. This is a unique unsigned integer value that references the worker thread that was created to manage the asynchronous client session. This function should only be called within an event handler that is invoked by a background task that has been started using a function such as **HttpAsyncGetFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskAbort, HttpTaskDone, HttpTaskResume, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetText Function

```
INT WINAPI HttpGetText(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **HttpGetText** function returns the contents of a text resource in a null-terminated string buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource that will be transferred to the local system. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A null-terminated string buffer which will contain the contents of the text file when the function returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetText** function is used to retrieve the contents of a text resource and return it in a null-terminated string buffer. Because binary data can include embedded null characters which would truncate the string, this function should only be used with text files or script output that is known to be textual. For example, it is safe to use this function when a resource returns HTML or XML data, but should not be used if it returns an image or executable file.

This function has been included as a convenience for applications that need to retrieve relatively small amounts of textual data and manipulate the contents as a string. If the Unicode version of this function is called, the text is automatically converted to a Unicode string. If the maximum amount of data being returned is unknown or the amount of text is very large, it is recommended that you use the **HttpGetTextEx** or **HttpGetFile** functions.

If the ANSI version of this function is called, the text returned will always be UTF-8 encoded. If you need to return the text using a different code page, use the **HttpGetTextEx** function which will allow you to specify a different code page used for the conversion.

If you use the **HttpGetFileSize** function to determine how large the string buffer should be prior to

calling this function, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **HttpGetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line characters.

HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = HttpGetText(hClient, lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetData, HttpGetFile, HttpGetTextEx, HttpPutData, HttpPutFile, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetThreadQueue Function

```
HQUEUE WINAPI HttpGetThreadQueue(
    DWORD dwThreadId
);
```

The **HttpGetThreadQueue** function returns a handle to the queue associated with the specified thread.

Parameters

dwThreadId

An unsigned integer value which specifies a unique identifier for the queue manager thread. If this value is zero, it will return the handle associated with the current thread.

Return Value

If the function succeeds, the return value is the handle to the queue. If the function fails, it will return INVAID_QUEUE. To get extended error information, call **HttpGetLastError**.

Remarks

All queued file transfers are handled by a queue manager which executes asynchronously in a background worker thread. This worker thread is created when the **HttpStartQueue** function is called and terminates when all files in the queue have been processed, or when the **HttpStopQueue** function is called. This function will return the queue handle associated with that worker thread.

The most common use of this function would be to obtain the handle for the queue from within a queue event handler. The application would pass in a value of zero as the thread ID and the function will return the handle to the queue which invoked the event notification.

If you call this function outside of an event handler and the ***dwThreadId*** parameter is zero, or if the queue is idle and not transferring files, this function will return INVALID_QUEUE. The thread ID is only valid while the queue manager is actively processing files in the queue.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetQueueStatus, HttpGetQueueThread, HttpStartQueue, HttpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetTimeout Function

```
INT WINAPI HttpGetTimeout(
   HCLIENT hClient
);
```

The **HttpGetTimeout** function returns the number of seconds the client will wait for a response from the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both the requests submitted and the responses returned by the server. The initial timeout value is set when establishing a connection and the default timeout period is 20 seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpConnect, HttpProxyConnect, HttpSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetTransferStatus Function

```
INT WINAPI HttpGetTransferStatus(
   HCLIENT hClient,
   LPHTTPTRANSFERSTATUS lpStatus
);
```

The **HttpGetTransferStatus** function returns information about the current data transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current data transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetTransferStatus** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the HTTPTRANSFERSTATUS structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **HttpGetTransferStatusEx** function which returns the size as a 64-bit value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

HttpEnableEvents, HttpGetTransferStatusEx, HttpRegisterEvent, HTTPTRANSFERSTATUS, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetTransferStatusEx Function

```
INT WINAPI HttpGetTransferStatusEx(
   HCLIENT hClient,
   LPHTTPTRANSFERSTATUSEX lpStatus
);
```

The **HttpGetTransferStatusEx** function returns information about the current data transfer in progress. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an HTTPTRANSFERSTATUSEX structure which contains information about the status of the current data transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetTransferStatusEx** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the ***dwTimeElapsed*** and ***dwTimeEstimated*** members of the HTTPTRANSFERSTATUSEX structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

HttpEnableEvents, HttpGetTransferStatus, HttpRegisterEvent, HTTPTRANSFERSTATUS, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpInitialize Function

```
BOOL WINAPI HttpInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **HttpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **HttpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **HttpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **HttpInitialize** or **HttpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnect, HttpDisconnect, HttpProxyConnect, HttpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpIsBlocking Function

```
BOOL WINAPI HttpIsBlocking(
   HCLIENT hClient
);
```

The **HttpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **HttpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCancel, HttpIsConnected, HttpIsReadable, HttpIsWritable, HttpRead, HttpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpIsConnected Function

```
BOOL WINAPI HttpIsConnected(
    HCLIENT hClient
);
```

The **HttpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpIsBlocking, HttpIsReadable, HttpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpIsReadable Function

```
BOOL WINAPI HttpIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **HttpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetStatus, HttpIsBlocking, HttpIsConnected, HttpIsWritable, HttpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpIsWritable Function

```
BOOL WINAPI HttpIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **HttpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetStatus, HttpIsBlocking, HttpIsConnected, HttpIsReadable, HttpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpNormalizePath Function

```
INT WINAPI HttpNormalizePath(
   LPCTSTR lpszResource,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **HttpNormalizePath** function normalizes a resource on the server returning a canonical path.

Parameters

lpszResource

A null-terminated string which specifies a resource on the server. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized resource path when the function returns. The buffer must be large enough to contain the entire path including the terminating null character. If this parameter is NULL it will be ignored and the function will return the length of the normalized path without copying any characters.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the function will return the length of the normalized path without copying any characters.

Return Value

This function returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized path but will not copy any characters. If the function fails, it will return HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function will accept a resource path on the server and return a normalized version of that path. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. For example, if the resource path provided is:

```
/text/..//project//./files/filename.pdf
```

This function would return the following as the normalized path:

```
/project/files/filename.pdf
```

The extraneous path delimiters are removed (the double slashes) and the reference to the parent directory is collapsed to produce a simple, uniform path. This can be useful both for display purposes and to normalize input provided by a user. This function will accept an HTTP URL and return only the normalized resource path for the URL. If you need to normalize a complete URL, use the **HttpNormalizeUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpNormalizeUrl, HttpValidateHostName HttpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpNormalizeUrl Function

```
INT WINAPI HttpNormalizeUrl(
   LPCTSTR lpszUrl,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **HttpNormalizeUrl** function normalizes the resource path and returns a canonical URL.

Parameters

lpszUrl

A null-terminated string which specifies an HTTP URL. This parameter cannot be NULL.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the function returns. The buffer must be large enough to contain the entire URL including the terminating null character. If this parameter is NULL it will be ignored and the function will return the length of the normalized URL without copying any characters.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer. If this value is zero, the ***lpszNormalized*** parameter is ignored and the function will return the length of the normalized URL without copying any characters.

Return Value

This function returns the number of characters copied into the string buffer, not including the terminating null character. If the ***lpszNormalized*** parameter is NULL, it will return the length of the normalized URL but will not copy any characters. If the function fails, it will return HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function will accept a standard HTTP or HTTPS URL and return a canonical version of that URL. Normalization will remove extraneous path delimiters and collapse the path if there are references to parent directories. If non-standard port numbers are included in the URL, they will be preserved. For example, if the follwing URL is provided to the function:

```
https://www.example.com:443/file/..//project/filename.pdf
```

It will return the following as the normalized URL:

```
https://www.example.com/project/filename.pdf
```

Because the port number specified in the original URL is the default port number for secure HTTP connections, it is removed. In addition, the path is collapsed and the extraneous path delimiters are removed.

This function is primarily intended to normalize URLs which may be entered by a user or read from a configuration file. It is not necessary to call this function prior to calling **HttpConnectUrl** because it will always be normalized internally prior to the connection being established. If you wish to check a URL to ensure it is valid and the server name can be resolved to an IP address, use the **HttpValidateUrl** function.

If you need to normalize a remote file path, use the **HttpNormalizePath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpNormalizePath, HttpValidateHostName HttpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpOpenFile Function

```
INT WINAPI HttpOpenFile(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   DWORD dwOffset
);
```

The **HttpOpenFile** function opens the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to open.

dwOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Use the **HttpCloseFile** function to close the file on the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCloseFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpOpenFileEx Function

```
INT WINAPI HttpOpenFileEx(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   ULARGE_INTEGER uiOffset
);
```

The **HttpOpenFileEx** function opens the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to open.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Use the **HttpCloseFile** function to close the file on the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCloseFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPatchData Function

```
INT WINAPI HttpPatchData(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszPatchData,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPatchData** function submits patch data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name that the patch data will be submitted to. Typically this is the name of a script on the server.

lpszPatchData

A null-terminated string that specifies the patch data that will be submitted to the server.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_PATCH_DEFAULT	The default post mode. The contents of the buffer will be submitted without encoding. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_PATCH_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

HTTP_PATCH_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPatchData** function is used to submit XML or JSON formatted patch data to a service, and then returns a copy of the response from the server into a local buffer. This function will not perform any encoding and will not automatically define the type of patch data being submitted. Your application is responsible for specifying the content type for the patch data, and ensuring that the XML or JSON data that is being submitted to the server is formatted correctly.

This function sends a PATCH command to the server, which is similar to a POST or PUT request. It is used to make partial updates to a resource, rather than creating or replacing it entirely. The format of the patch data is specific to the service being used. If the resource being patched does not exist, the behavior is defined by the server. If enough information is provided, it may choose to create the resource just as if a PUT command was used, or it may return an error.

Your application should use the **HttpSetRequestHeader** function to define the Content-Type header prior to calling the **HttpPatchData** function. One of the most common formats used is the JSON Merge Patch which is defined in RFC 7396. The value for the Content-Type header for this patch format is "application/merge-patch+json". Refer to your service API documentation to determine what patch formats are acceptable, along with any additional header values that must be defined.

The function may be used in one of two ways, depending on the needs of the application. The first approach is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the ***lpvResult*** parameter will point to the buffer that was allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second approach is to have the ***lpvResult*** parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the ***lpcbResult*** parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the PATCH request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

When encountering a server error during a request, the **HttpPatchData** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the PATCH request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_PATCH_ERRORDATA option. When this

option is enabled, the behavior of **HttpPatchData** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification can be enabled by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_PATCH_CONVERT option is used

nResult = HttpPatchData(hClient,
                        lpszResource,
                        lpszPatchData,
                        &hgblBuffer,
                        &cbBuffer,
                        HTTP_PATCH_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetData, HttpGetTransferStatus, HttpPostData, HttpPostJson, HttpPostXml, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPostData Function

```
INT WINAPI HttpPostData(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCVOID lpvBuffer,
   DWORD cbBuffer,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPostData** function submits the contents of the specified buffer to a script on the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource that the data will be posted to on the server. Typically this is the name of a script that will be executed. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

The data that will be provided to the script. This parameter may be NULL if the script does not require any additional data from the client.

cbBuffer

The number of bytes to copy from the buffer. If this ***lpvBuffer*** parameter is NULL, this value should be zero.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character

	sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_MULTIPART	The contents of the buffer being sent to the server consists of multipart form data and will be sent as-is without any encoding.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostData** function is used to submit data to a script that executes on the server and then copy the output from that script into a local buffer. If you are submitting XML or JSON formatted data to the server, it is recommended that you use the **HttpPostXml** or **HttpPostJson** functions instead to ensure that the correct content type and encoding is automatically selected.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the *lpcbResult* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

When encountering a server error during a request, the **HttpPostData** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **HttpPostData** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call

HttpGetResultCode to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification can be enabled by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the response in block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = HttpPostData(hClient,
                       lpszResource,
                       lpParameters,
                       cbParameters,
                       &hgblBuffer,
                       &cbBuffer,
                       HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetData, HttpGetTransferStatus, HttpPostJson, HttpPostXml, HttpPutData, HttpRegisterEvent, HttpSubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPostFile Function

```
INT WINAPI HttpPostFile(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LPCTSTR lpszResource,
   LPCTSTR lpszFieldName,
   DWORD dwOptions,
   DWORD dwReserved
);
```

The **HttpPostFile** function posts the contents of the specified file to a script executed on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszResource

A null-terminated string that specifies the resource on the server that the file data will be posted to. Typically this is the name of a script that is responsible for processing and storing the file data.

lpszFieldName

A null-terminated string that corresponds to the form field name that the script expects. If this parameter is NULL or an empty string, a default field name of "File1" is used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostFile** function is similar to the **HttpPutFile** function in that it can be used to upload the contents of a local file to a server. However, instead of using the PUT command, the POST command is used to send the file data to a script that is executed on the server. This method has the advantage of not requiring any special configuration settings on the server, however it does require that the script be able to process multipart/form-data as defined in RFC 2388.

To support uploading files from a form on a webpage, the FILE input type is used along with the

action that specifies the script that will accept the file data and process it. For example, the HTML code could look like this:

```
<form action="/cgi-bin/upload.cgi" method="post" enctype="multipart/form-
data">
<input type="file" name="datafile" size="20">
<input type="submit">
</form>
```

In this example, the script /cgi-bin/upload.cgi is responsible for processing the file data that is posted by the client, and the form field name "datafile" is used. The user can select a file, and when the Submit button is clicked, the file data is posted to the script. To simulate this using the **HttpPostFile** function, the *lpszFileName* parameter should be set to the name of the local file that will be posted to the server. The *lpszResource* parameter should be the name of the script, in this case "/cgi-bin/upload.cgi". The *lpszFieldName* parameter should be specified as the string "datafile" to match the name of the field used by the form.

Note that the **HttpPostFile** function always submits the file contents as multipart/form-data with the content type set to application/octet-stream. The script that accepts the posted data must be able to parse the multipart header block and correctly process 8-bit data. If the script assumes that the data will be posted using a specific encoding type such as base64 then the file data may not be accepted or may be corrupted by the script.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetData, HttpGetFile, HttpGetTransferStatus, HttpPostData, HttpPutData, HttpPutFile, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPostJson Function

```
INT WINAPI HttpPostJson(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszJsonData,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPostJson** function submits JSON structured data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name the JSON data will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszJsonData

A null-terminated string which specifies the JSON structured text to be submitted to the server. If the Unicode version of this function is called, the text will sent as a UTF-8 encoded string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function

	will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostJson** function is used to submit JSON structured data to a REST API endpoint or script and then returns the output in a local buffer. This function automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure it is well-formed. Your application is responsible for ensuring the JSON data being submitted to the server is formatted correctly.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the *lpcbResult* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "application/json" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

When encountering a server error during a request, the **HttpPostJson** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **HttpPostJson** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the JSON payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the

server to respond.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD cbBuffer = 0;

// Store the response in a block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter.

nResult = HttpPostJson(hClient,
                       lpszResource,
                       lpszJsonData,
                       &hgblBuffer,
                       &cbBuffer,
                       HTTP_POST_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpGetResultCode, HttpGetText, HttpPostText, HttpPostXml, HttpPutText, HttpSetRequestHeader, HttpSubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPostText Function

```
INT WINAPI HttpPostText(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszTextData,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPostText** function submits plain text data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name the text will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszTextData

A null-terminated string which specifies the text to be submitted to the server. If the Unicode version of this function is called, the text will sent as a UTF-8 encoded string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function

	will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostText** function is used to submit plain text to a REST API endpoint or script and then returns the output in a local buffer. This function can be used to submit any kind of text payload to the server, and automatically sets a default content type without any encoding, ensuring the text is submitted exactly as specified. If the Unicode version of this function is called, the text will be converted to UTF-8 prior to being submitted to the server. The endpoint or script on the server side must be capable of processing UTF-8 encoded text correctly or you may experience unexpected results.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the generic "text/plain" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type. Your application should ensure any content type you specify is valid for the contents of the payload; the content type is not checked against the text you submit to the server.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the *lpcbResult* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

If you are using the Unicode version of this function, the data returned in the *lpvResult* buffer will always contain bytes and not wide (16-bit) characters. It will typically consist of UTF-8 encoded text; however, depending on the request, the server may return binary data. The value of the Content-Type header returned by the server as part of its response will indicate what type of data the payload contains. If you expect the server to return text, you can use the **MultiByteToWideChar** function to convert the response to a Unicode string, or you can use the **HttpPostTextEx** function, which can handle the text conversion for you.

When encountering a server error during a request, the **HttpPostText** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic

messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **HttpPostText** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the text payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the server to respond.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD cbBuffer = 0;

// Store the response in a block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = HttpPostText(hClient,
                       lpszResource,
                       lpszTextData,
                       &hgblBuffer,
                       &cbBuffer,
                       HTTP_POST_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpGetResultCode, HttpGetText, HttpPostJson, HttpPostTextEx, HttpPostXml, HttpSetRequestHeader, HttpSubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPostTextEx Function

```
INT WINAPI HttpPostTextEx(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszTextData,
   LPTSTR lpszResult,
   LPDWORD lpcchResult,
   UINT nCodePage,
   DWORD dwOptions
);
```

The **HttpPostTextEx** function submits plain text data to the server and returns the result in a string provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name the text will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszTextData

A null-terminated string which specifies the text to be submitted to the server. If the Unicode version of this function is called, the text will sent as a UTF-8 encoded string.

lpszResult

A null-terminated string buffer which will contain the text response returned by the server. This parameter cannot be NULL.

lpcchResult

A pointer to an unsigned integer which should be initialized to the maximum number of characters that can be copied to the string specified by the ***lpszResult*** parameter. When the function returns, this value will be updated with the number of characters returned by the server, not including the terminating null character.

nCodePage

An unsigned integer that specifies the ANSI code page which should be used to convert the text returned by the server. A value of zero specifies the default system code page should be used with the conversion. In most cases, applications should use CP_UTF8 as the code page. See the remarks below for more information.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option

	does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostTextEx** function is used to submit plain text to a REST API endpoint or script and then returns the output in string provided by the caller. This function can be used to submit any kind of text payload to the server, and automatically sets a default content type, ensuring the text is submitted exactly as specified. If the Unicode version of this function is called, the text will be converted to UTF-8 prior to being submitted to the server. The endpoint or script on the server side must be capable of processing UTF-8 encoded text correctly or you may experience unexpected results.

This function differs from the **HttpPostText** function because it expects the response from the server to contain only text and will automatically convert the response to Unicode using the code page specified by the *nCodePage* parameter. If the Unicode version of this function is called, the text returned by the server will be converted to UTF-16 text and returned as a null-terminated string. If the ANSI version of this function is called, the text returned by the server will be converted to UTF-8 text and returned in the string buffer. This function should never be used to submit a request when the response from the server may contain binary data.

In most cases, you should specify the *nCodePage* parameter as **CP_UTF8** because this is what most servers use as their default character set. If the server is returning text using a character set other than UTF-8, specifying a code page enables your application to automatically normalize that text to Unicode. It is important to note the text returned by this function will always be either UTF-16 or UTF-8 Unicode to ensure there is no data loss during the conversion process. If you have a specific need to return a localized ANSI string, use **HttpPostTextExW** and then convert the Unicode string with **WideCharToMultiByte** using **CP_ACP** as the code page. This can potentially result in data loss if the current ANSI code page cannot represent all of the Unicode text returned by the server.

The string buffer specified by *lpszResult* should be large enough to contain the entire response from the server, including the terminating null character. If the buffer is not large enough, the response returned by the server will be truncated. If you cannot estimate what the largest reasonable buffer size should be, it's recommended you use the **HttpPostText** function instead which can return an HGLOBAL handle rather than a fixed-length buffer.

It is important for your application to initialize the value of the *lpcchResult* parameter correctly. This parameter must be passed by reference and if not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the generic "text/plain" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers

require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type. Your application should ensure any content type you specify is valid for the contents of the payload; the content type is not checked against the text you submit to the server.

When encountering a server error during a request, the **HttpPostText** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **HttpPostText** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the text payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the server to respond.

Example

```
TCHAR szResponse[MAXBUFSIZE];
DWORD cchResponse = MAXBUFSIZE;

// Submit the request to the server and return the response in a
// string buffer. End-of-line characters will be automatically
// converted to use Windows conventions. Note that cchResponse
// must be initialized with the maximum string length and will
// be updated to contain the actual string length when the function
// returns.

nResult = HttpPostTextEx(hClient,
                         lpszResource,
                         lpszTextData,
                         szResponse,
                         &cchResponse,
                         HTTP_POST_ERRORDATA);

if (nResult != HTTP_ERROR)
{
    // Because we have used the HTTP_POST_ERRORDATA option, the
    // function will not return an error if the server returns
    // a status code in the 400 or 500 range. In this case, the
    // the response will typically contain more information
    // about the cause of the failure.

    INT nCode = HttpGetResultCode(hClient);

    if (nCode >= 200 && nCode <= 299)
    {
        // The server has indicated the request was successful
```

```
    }
    else if (nCode >= 400)
    {
        // There was some problem with the request; check the
        // szResponse string for more information
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpGetResultCode, HttpGetText, HttpPostJson, HttpPostText, HttpPostXml, HttpSetRequestHeader, HttpSubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPostXml Function

```
INT WINAPI HttpPostXml(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszXmlData,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPostXml** function submits XML structured data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource name the XML data will be submitted to. Typically this is the name of a REST API endpoint or script on the server.

lpszXmlData

A null-terminated string which specifies the XML structured text to be submitted to the server. If the Unicode version of this function is called, the text will sent as a UTF-8 encoded string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the number of bytes of data returned by the server.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_POST_DEFAULT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. This option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function

	will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostXml** function is used to submit XML structured data to a REST API endpoint or script and then returns the output in a local buffer. This function automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure it is well-formed. Your application is responsible for ensuring the XML data being submitted to the server is formatted correctly.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is important for your application to initialize the value of the *lpcbResult* parameter correctly. This parameter must be passed by reference and if it is not initialized properly it can cause unexpected behavior or corrupt the process heap. If an error occurs when issuing the POST request to the server, the variable passed to this function may have a value of zero when the function returns, indicating no data has been returned to the caller.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "text/xml" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

When encountering a server error during a request, the **HttpPostXml** function normally returns HTTP_ERROR, and no data is copied into the caller-provided buffer. The last error code for the current thread is updated to reflect the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted text containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the HTTP_POST_ERRORDATA option. When this option is enabled, the behavior of **HttpPostXml** changes; it does not return HTTP_ERROR for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the POST completes, a timeout occurs or the operation is canceled. To prevent the request from blocking the main application thread, create a background worker thread to submit the XML payload. Connections established in different threads will not interfere with one another and will not cause the main thread to block while it waits for the

server to respond.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD cbBuffer = 0;

// Store the response in a block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter.

nResult = HttpPostXml(hClient,
                      lpszResource,
                      lpszXmlData,
                      &hgblBuffer,
                      &cbBuffer,
                      HTTP_POST_ERRORDATA);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Because we have used the HTTP_POST_ERRORDATA option, the
    // function will not return an error if the server returns
    // a status code in the 400 or 500 range. In this case, the
    // the response will typically contain more information
    // about the cause of the failure.

    INT nCode = HttpGetResultCode(hClient);

    if (nCode >= 200 && nCode <= 299)
    {
        // The server has indicated the request was successful
    }
    else if (nCode >= 400)
    {
        // There was some problem with the request; check the
        // contents of lpBuffer for more information
    }

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetResponseHeader, HttpGetResultCode, HttpGetText, HttpPostJson, HttpPostText, HttpPutText,

HttpSetRequestHeader, HttpSubmitRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpProxyConnect Function

```
HCLIENT WINAPI HttpProxyConnect(
   UINT nProxyType,
   LPCTSTR lpszProxyHost,
   UINT nProxyPort,
   LPCTSTR lpszProxyUser,
   LPCTSTR lpszProxyPassword,
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   DWORD dwVersion,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **HttpProxyConnect** function is used to establish a connection through a proxy server.

Parameters

nProxyType

An unsigned integer which specifies the type of proxy that the client is connecting to. The supported proxy server types are as follows:

Value	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the ***dwOptions*** parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server settings configured in Windows.

lpszProxyHost

A null-terminated string which specifies the proxy server host name or IP address. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

nProxyPort

The port number that the proxy server is listening for connections on. A value of zero specifies that the default port number 80 should be used. Note that in most cases, a proxy server is not configured to use the default port. This argument is ignored if the proxy type is set to

HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

lpszProxyUser

A null-terminated string which specifies the user name that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszProxyPassword

A null-terminated string which specifies the password that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszRemoteHost

A null-terminated string which specifies the name of the server to connect to through the proxy server. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 80 should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.

HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. This option will always establish a secure connection with the server. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_HTTP3	This option specifies the client should attempt a HTTP/3 (QUIC) connection with the server. If a connection cannot be established using HTTP/3 the client will attempt a connection using an earlier version of the protocol. You should only use this option if you know the server supports the QUIC protocol, otherwise requests will be delayed as a result of the fallback to attempting a HTTP/2 connection. The value of the ***dwVersion*** parameter will be ignored when this option is used.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the client will default to using TLS 1.2 or later. If HTTP_PORT_DEFAULT is specified as the remote port, this option will automatically change the port number to use HTTP_PORT_SECURE instead.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that

	use RC4, MD5 and SHA1. Although this option may permit connections with servers which do not support current standards, avoid using it unless absolutely necessary. Some servers may reject the connection if the client offers to use TLS 1.0, even when TLS 1.2 or later is also supported.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create a version value and the following values are defined:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.
HTTP_VERSION_09	The client should use the original one-line protocol which includes no version number and no request header block. This version has been deprecated and should only be used with legacy servers which do not support the protocol standard. This version of the protocol does not support virtual hosts and is not supported by most modern web services.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.

HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. If the client or server does not support HTTP/2, the client will automatically attempt to use an earlier version of the protocol.
HTTP_VERSION_30	The client should use the HTTP/3 protocol standard defined in RFC 9114. This protocol version is similar to earlier versions of the protocol in terms the request methods and server responses, but the underlying transport mechanism is significantly different. Instead of using TCP, this version of the protocol uses multiplexed UDP connections designed to reduce the overhead of TCP while maintaining reliability. If the client or server does not support HTTP/3, the client will automatically attempt to use an earlier version of the protocol.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to the client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **HttpProxyConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

If the HTTP_PROXY_WINDOWS proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related parameters will be ignored and the function will establish a connection directly to the server.

The username and password information is only used when connecting to a server which supports version 1.1 or later of the protocol.

Most applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If the HTTP_OPTION_KEEPALIVE option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the function will fail and return INVALID_CLIENT.

You should only attempt to use HTTP/3 if you know the server supports that version of the protocol. If the server does not support HTTP/3, the client will wait for the timeout period to elapse to determine the server is not responding to the HTTP/3 request and will then attempt to use HTTP/2. If you specify HTTP/3 as the protocol version and you are experiencing long delays for requests the complete, the most likely reason is the server does not support the QUIC protocol and every request is falling back to use HTTP/2 instead. The minimum required platform for HTTP/3 support is Windows 11 21H2 and later builds or Windows Server 2022 or later.

The handle returned by this function is initially attached to the thread that created it. From that point on, until it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnect, HttpConnectUrl, HttpDisconnect, HttpInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPutData Function

```
INT WINAPI HttpPutData(
   HCLIENT hClient,
   LPCTSTR lpszRemoteFile,
   LPCVOID lpvBuffer,
   DWORD dwLength,
   DWORD dwReserved
);
```

The **HttpPutData** function copies the contents of the specified buffer and stores it in a file on the server.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A buffer which will contain the data to be transferred from the client and stored in a file on the server.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutData** function is used to store the contents of the specified buffer in a file on the server. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** function prior to uploading the data.

This function is designed for uploading the contents of a buffer to be stored as a file on the server. If you need to use the PUT command to as part of a RESTful API where the PUT command is used to submit data to the server, use the **HttpPutDataEx** function. That function will return the success or failure responses back to the client.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetData, HttpGetFile, HttpGetTransferStatus, HttpPostData, HttpPutDataEx, HttpPutFile, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPutDataEx Function

```
INT WINAPI HttpPutDataEx(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCVOID lpvBuffer,
   DWORD cbBuffer,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPutDataEx** function submits the contents of the specified buffer to the server using the PUT command and returns the server's response to the command.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string which specifies the resource path on the server. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A buffer which will contain the data to be transferred from the client and submitted to the server. This parameter may not be NULL.

cbBuffer

The number of bytes to copy from the buffer. This value must be larger than zero.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_PUT_DEFAULT	The default mode. The contents of the buffer are sent unmodified and the data returned by the server is copied to the result buffer exactly as it is returned from the server. This option should be used if the server is expected to return binary data in the response to the PUT request.
HTTP_PUT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual

	carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_PUT_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutDataEx** function is used to submit data to the server using the PUT command. This is typically used with RESTful APIs and is similar in functionality to the POST command. However, the data that is submitted will be sent to the server as-is and will not be encoded.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the response from the server. In this case, the ***lpvResult*** parameter will point to the buffer allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second method is to have the ***lpvResult*** parameter point to a global memory handle which will reference the data when the function returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

If the Unicode version of this function is called and the server returns a text response to the PUT request, it is the application's responsibility to convert the contents of the ***lpvResult*** buffer to UTF-16 text. If the server returns text, most likely it will use UTF-8 encoding.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "application/octet-stream" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_PUT_ERRORDATA option which causes the error message to be returned in the ***lpvResult*** buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the data transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a data transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgblResponse = (HGLOBAL)NULL;
LPBYTE lpResponse = (LPBYTE)NULL;
DWORD cbResponse = 0;

// Store the response into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the response from
// the server is textual, the HTTP_PUT_CONVERT option is used

nResult = HttpPutDataEx(hClient,
                        lpszResource,
                        lpRequest,
                        cbRequest,
                        &hgblResponse,
                        &cbResponse,
                        HTTP_PUT_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // response from the server
    lpResponse = (LPBYTE)GlobalLock(hgblResponse);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblResponse);
    GlobalFree(hgblResponse);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetData, HttpGetFile, HttpGetTransferStatus, HttpPostData, HttpPutData, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPutFile Function

```
INT WINAPI HttpPutFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
   DWORD dwOffset
);
```

The **HttpPutFile** function transfers the specified file on the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local system.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

dwOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutFile** function is used to transfer a file from the local system to a server. Not all servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** function prior to uploading the file.

To upload large files that are over 4GB, use the **HttpPutFileEx** function.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically

fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetFile, HttpGetTransferStatus, HttpPostData, HttpPostFile, HttpPutData, HttpPutFileEx, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPutFileEx Function

```
INT WINAPI HttpPutFileEx(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwOptions
   ULARGE_INTEGER uiOffset
);
```

The **HttpPutFileEx** function transfers the specified file on the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Value	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutFileEx** function is used to transfer a file from the local system to a server. Not all servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** function prior to uploading the file.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be

enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetFileEx, HttpGetTransferStatusEx, HttpPostData, HttpPostFile, HttpPutData, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPutText Function

```
INT WINAPI HttpPutText(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszBuffer
);
```

The **HttpPutText** function creates a text file on the server using the contents of a string buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string that specifies the resource on the server which will be created or overwritten. This string may specify a valid URL for the current server.

lpszBuffer

A null-terminated string that contains the text which will be submitted to the server .

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutText** function is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** command prior to uploading the data.

If the Unicode version of this function is called, the string will be automatically converted to UTF-8 and then uploaded to the server. If you wish to upload UTF-16 Unicode or ANSI text to the server, you must use the **HttpPutData** function. This function should never be used to upload binary data.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetText, HttpGetTransferStatus, HttpPutData, HttpPutFile, HttpRegisterEvent

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpPutTextEx Function

```
INT WINAPI HttpPutTextEx(
   HCLIENT hClient,
   LPCTSTR lpszResource,
   LPCTSTR lpszBuffer,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpPutTextEx** function submits the contents of a string buffer to the server using the PUT command and returns the server's response to the command.

Parameters

hClient

Handle to the client session.

lpszResource

A null-terminated string which specifies the resource path on the server. This string may specify a valid URL for the current server.

lpszBuffer

A null-terminated string that contains the text that will be submitted to the server. This parameter cannot be NULL and may not specify an empty string.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResult*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_PUT_DEFAULT	The default mode. automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_PUT_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutTextEx** function is used to submit the contents of a null-terminated string to the server using the PUT command. This is typically used with RESTful APIs and is similar in functionality to the POST command. However, the text that is submitted will not be encoded in the same way that the **HttpPostData** function submits data.

If the Unicode version of this function is called, the text will be converted to UTF-8 and then submitted to the server. If you wish to submit the contents of the string as UTF-16 Unicode text, or the payload contains binary data, you must use the **HttpPutDataEx** function. This function should never be used to upload binary data. If the server returns a text response and the Unicode version of this function is called, it is the application's responsibility to convert the contents of the *lpvResult* buffer to UTF-16 text. If the server returns text, most likely it will use UTF-8 encoding.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "text/plain" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_PUT_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the data transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a data transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnableEvents, HttpGetText, HttpGetTransferStatus, HttpPutDataEx, HttpPutFile, HttpPutText, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRead Function

```
INT WINAPI HttpRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **HttpRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When **HttpRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpIsBlocking, HttpIsReadable, HttpIsWritable, HttpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRegisterEvent Function

```
INT WINAPI HttpRegisterEvent(
   HCLIENT hClient,
   UINT nEventId,
   HTTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **HttpRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
HTTP_EVENT_CONNECT	The connection to the server has completed.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a

	transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event occurs when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **HttpRegisterEvent** function associates a callback function with a specific event. The event handler is an **HttpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a file is being uploaded or downloaded. The HTTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one HTTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpDisableEvents, HttpEnableEvents, HttpEventProc, HttpFreezeEvents, HttpGetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRemoveQueuedFile Function

```
BOOL WINAPI HttpRemoveQueuedFile(
    HQUEUE hQueue,
    DWORD dwFileId
);
```

The **HttpRemoveQueuedFile** removes the specified file from the queue.

Parameters

hQueue

A handle to a file transfer queue.

dwFileId

An unsigned integer value which uniquely identifies the file in the queue. This value cannot be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

Files can only be removed from the queue when in an idle or paused state. If you attempt to remove a file while the queue manager is in the process of uploading or download files, the function will fail.

The file identifier can be obtained by several different functions, including **HttpEnumQueuedFiles** and **HttpGetQueueStatus,** which provides information about the current file being processed in the queue. Do not make any assumptions about the value of the identifier. Although the value is guaranteed to be unique for the specified queue, it is not guaranteed that file identifiers will be assigned in sequential order.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpAddQueuedFile, HttpClearQueue, HttpEnumQueuedFiles, HttpGetQueuedFile, HttpGetQueueStatus, HttpSuspendQueue, HttpResetQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpResetQueue Function

```
BOOL WINAPI HttpResetQueue(
    HQUEUE hQueue,
    DWORD dwOptions
);
```

The **HttpResetQueue** function resets the state of all file transfers in the current queue.

Parameters

hQueue

A handle to a file transfer queue.

dwOptions

An integer value which specifies which files should be reset in the queue. It may be one of the following values:

Value	Description
HTTP_QUEUE_RESET_ALL	All files in the queue should be reset to their initial state.
HTTP_QUEUE_RESET_COMPLETED	All files in the queue which have been successfully transferred will be reset to their initial state.
HTTP_QUEUE_RESET_FAILED	All files in the queue which were not transferred will be reset to their initial state.
HTTP_QUEUE_RESET_CANCELED	All files in the queue which were canceled will be reset to their initial state.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

When the **HttpStartQueue** function is called multiple times using the same queue handle, it will not attempt to transfer files which have already been successfully copied, and it will not attempt to re-transfer files if the previous transfer failed. This function resets queued files back to their initial state, prior to when the queue manager attempted to perform the transfer. The most common use would be to call the function with the HTTP_QUEUE_RESET_FAILED mode and then call **HttpStartQueue** to retry failed or canceled file transfers.

Because canceled file transfers are also considered failed transfers, specifying HTTP_QUEUE_RESET_FAILED will reset queued files which either encountered an error during the transfer or were explicitly canceled by calling the **HttpCancelQueuedFile** function. The HTTP_QUEUE_RESET_CANCELED option will only reset the state of queued file transfers which were canceled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtpv12.lib

See Also

HttpCancelQueuedFile, HttpGetQueuedFile, HttpResumeQueue, HttpStartQueue, HttpStopQueue, HttpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpResumeQueue Function

```
BOOL WINAPI HttpResumeQueue(
    HQUEUE hQueue
);
```

The **HttpResumeQueue** function resumes transfers in the queue.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function signals the queue manager to resume processing files in the transfer queue after the **HttpSuspendQueue** function has been called. This function will fail if the queue is in an idle (stopped) state, and will be ignored if the queue manager is already transferring files. The **HttpGetQueueStatus** function can be called to obtain the current status of the queue.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetQueueStatus, HttpResetQueue, HttpStartQueue, HttpStopQueue, HttpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetBearerToken Function

```
INT WINAPI HttpSetBearerToken(
   HCLIENT hClient,
   LPTSTR lpszBearerToken
);
```

The **HttpSetBearerToken** function sets the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

hClient

Handle to the client session.

lpszBearerToken

A null-terminated string buffer which contains the bearer token used to authorize client requests. If this parameter is NULL or a zero-length string, the current bearer token will be cleared and no client authentication will be performed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Using OAuth 2.0 requires you to understand the process of how to request the bearer (access) token. Obtaining a bearer token requires registering your application with the web service provider, getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access the service.

Your application should not store a bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticate, HttpGetBearerToken, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetContentType Function

```
BOOL WINAPI HttpSetContentType(
   HCLIENT hClient,
   LPCTSTR lpszContentType
);
```

The **HttpSetContentType** function sets the content type for the next request.

Parameters

hClient

Handle to the client session.

lpszContentType

A null-terminated string which contains the MIME content type of the data being submitted to the web server. If this parameter is NULL or an empty string, the current type will be cleared and the client will use a default type based on the request and content encoding.

Return Value

If the function succeeds, it will return a non-zero value. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

The content type string must use the standard IANA format which consists of a type and subtype separated by a forward slash. Some common content types are:

Content Type	Description
`text/plain`	The request payload is unstructured text in a human readable format. This content type serves a default for textual data which does not explicitly have another standard subtype.
`text/csv`	The request payload is in a comma-separated value (CSV) text format.
`application/octet-stream`	The request payload contains binary data. This content type serves a default for binary data which does not explicitly have another standard subtype.
`application/json`	The request payload is standard JSON formatted text. This is the default content type used when submitting JSON requests, however some services may expect the client to use the non-standard `text/x-json` type.
`application/xml`	The request payload is standard XML formatted text. This is the default content type used when submitting XML requests, however some servers may expect the client use the deprecated `text/xml` type. The standard content type for XML is defined in RFC 7303, however most services will recognize both types as valid.

If a content type is not explicitly specified with a request, an appropriate default content type will be automatically selected. For example, the **HttpPostJson** function will set the content type to **application/json** by default. This function can be used to override that default value.

This function will validate the content type, but does not validate the subtype. Unregistered subtypes should always be prefixed with "x-" which designates it as experimental. Some commonly used types also use this designation, such as **text/x-vcalendar** for the vCalendar format.

If your application needs to set the content type to a non-standard value for a proprietary API, use the **HttpSetRequestHeader** function to set the Content-Type header value. That function will not perform any validation on the header value. The contents of the payload will not be checked to ensure the content type is valid. It is the responsibility of the application to ensure the correct content type is used when submitting the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetContentType, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetCookie Function

```
BOOL WINAPI HttpSetCookie(
   HCLIENT hClient,
   LPCTSTR lpszCookieName,
   LPCTSTR lpszCookieValue
);
```

The **HttpSetCookie** function sends the specified cookie to the server when a resource is requested.

Parameters

hClient

Handle to the client session.

lpszCookieName

Pointer to a string which specifies the name of the cookie that will be sent to the server when the next resource is requested.

lpszCookieValue

Pointer to a string which specifies the value of the cookie. To delete a cookie that has been previously set, this parameter should be NULL or point to an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetCookie** function submits the cookie name and value to the server when a resource is requested or data is posted to a script. For more information about cookies and how they are used, refer to the **HttpGetCookie** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpClearHeaders, HttpGetCookie, HttpGetFirstCookie, HttpGetNextCookie, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetDefaultUserAgent Function

```
INT WINAPI HttpSetDefaultUserAgent(
    LPCTSTR lpszUserAgent
);
```

The **HttpSetDefaultUserAgent** function sets the default user agent string which is included with all requests.

Parameters

lpszUserAgent

A null-terminated string which specifies the new default user agent string. If this parameter is NULL or an empty string, the client will revert to using the default user agent string defined when the library was first initialized.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetDefaultUserAgent** function changes the default user agent string which is included with each request made by the client. This default value is global and changing it will change the default user agent for all client sessions in all threads created by the process.

The user agent string should use a format with the product name and version number separated by a slash. There can be multiple products listed in the user agent string separated by spaces, with additional optional information enclosed in parenthesis. For example, a valid user agent string for an application named "MyProgram" could look like this:

```
MyProgram/1.0 (Win32)
```

Some web services check the value of the user agent string to determine if a compatible client is being used to issue the request. The default value used by SocketTools is designed to emulate a common browser, but some services may require you change the user agent to use a specific value or include certain product names and versions.

To change the user agent string for a specific client session, use the **HttpSetRequestHeader** function to set a value for the `User-Agent` header. To get the current value of the default user agent string, use the **HttpGetDefaultUserAgent** function.

To prevent any user agent string from being included with a request, include the HTTP_OPTION_NOUSERAGENT option when connecting to the server or use **HttpSetOption** to enable that option after a connection has been established.

Using the HTTP_OPTION_NOUSERAGENT option to prevent sending a User-Agent header does not make requests anonymous. Servers may still identify clients based on IP address information, authentication credentials, request patterns or other HTTP headers included with the request.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetDefaultUserAgent, HttpSetOption, HttpSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetDefaultZoneId Function

```
BOOL WINAPI HttpSetDefaultZoneId(
    HCLIENT hClient,
    DWORD dwZoneId
);
```

The **HttpSetDefaultZoneId** function sets the default security zone for downloaded files.

Parameters

hClient

Handle to the client session.

dwZoneId

An unsigned integer which specifies the default security zone identifier. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it. This typically indicates that the file was created locally or has not been marked by Windows as originating from another zone. This is the default behavior if the application does not explicitly set a default security zone.
SECURITY_ZONE_LOCAL	The file is considered to be part of the **Local Machine** zone, which includes files stored on the user's local system or removable media. This zone has the least restrictive security settings, and files are typically allowed to run without prompting or limitation. ***This zone should not be assigned to downloaded files***, as doing so can bypass security mechanisms designed to protect against potentially harmful content. Only use this setting for files that are created locally by the application itself and are known to be safe. Misuse of this setting can result in a significant security risk, especially if the file contains scripts, executables, or other active content.
SECURITY_ZONE_INTRANET	The file originates from the **Local Intranet** zone, which includes content served from internal network resources, such as shared drives, file servers, and internal web applications. This zone typically has fewer security restrictions than the Internet zone but more than the Local Machine zone. It is commonly used in enterprise environments where network resources are managed and considered relatively safe. ***This zone should not be used for files downloaded from the public Internet***, as doing so may incorrectly signal that the file is trusted within the organization. Assign this zone only to files that are retrieved from authenticated internal sources or known secure locations on the local network.
SECURITY_ZONE_TRUSTED	The file is from a site or source explicitly added to the

	Trusted Sites zone in Internet Explorer or Windows security settings. This zone is intended for domains or locations that the user or system administrator considers to be safe, such as internal applications hosted on the public Internet or third-party services with a strong reputation for safety. Files marked with this zone typically bypass certain security checks, and applications may allow active content (such as scripts or macros) to run without prompting. Use caution when assigning this zone to downloaded files, as it may override protections that would otherwise apply to untrusted content.
SECURITY_ZONE_INTERNET	The file was downloaded from the **Internet** zone, which represents untrusted or unknown sources outside the local network. This is the standard zone identifier automatically applied by web browsers, email clients, and many applications when a file is downloaded from a public website or external server. Windows uses this zone to enforce security restrictions, such as prompting the user before executing scripts, blocking active content, or displaying a warning when the file is opened. This is typically the appropriate zone to assign for any file obtained over HTTP from an external site. Assigning this zone helps ensure that users are notified before opening potentially unsafe content, in accordance with Windows Attachment Manager and Group Policy settings.
SECURITY_ZONE_RESTRICTED	The file is associated with the **Restricted Sites** zone, which enforces the strictest security policies. This zone is used for content that is explicitly marked as untrusted or potentially harmful. Files tagged with this zone may be blocked from execution, prevent active content from running, or trigger warnings when accessed. Assigning this zone can be useful for sandboxing unverified downloads or limiting the risk from files obtained from questionable sources. However, it may also cause legitimate files to fail to open or operate with reduced functionality depending on system policy and application behavior.

Return Value

If the function succeeds, the return value is non-zero. If the function fails it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetDefaultZoneId** function sets the default security zone identifier that will be applied to all files downloaded during the client session. This setting controls how Windows treats downloaded files with respect to security prompts and content execution restrictions. Once the default zone ID is set, it will be automatically applied to all subsequent file downloads performed by the client session. This eliminates the need to manually set the security zone for each individual file, ensuring consistent zone tagging across the session.

When is specified, no alternate data stream is created. This

`SECURITY_ZONE_NONE` `Zone.Identifier`

causes the file to be treated as if it originated from the local system, and Windows will not apply zone-based security restrictions. This is the default behavior which maintains backwards compatibility with previous versions of the library that did not create the security zone metadata for files.

The zone ID metadata is interpreted by Windows components such as **Attachment Execution Services** and **SmartScreen** to determine whether the user should be warned when opening a file. In enterprise environments, Group Policy settings may alter how different zone IDs are handled, such as blocking scripts, macros, or unsigned executables. Developers should test zone assignment behavior in target environments when distributing files to end users.

Note that this function does not retroactively modify existing files. The default zone identifier is applied only at the time a file is downloaded using functions such as **HttpDownloadFile**.

If your application is distributed in a controlled environment, such as a corporate network with endpoint protection or user access controls, consult your organization's security policy before modifying default zone behavior. Improper zone tagging can impact file access, generate security prompts, or interfere with digital signature enforcement.

Be aware that some file operations (such as extracting from a ZIP archive, or copying files to non-NTFS volumes or across certain network shares) may strip the `Zone.Identifier` metadata. Developers should not rely solely on zone tagging as a persistent trust mechanism outside of the initial download context.

To get the security zone identifier for a local file, use the **HttpGetFileZoneInfo** function. To change or remove the security zone information for a file, use the **HttpSetFileZoneInfo** function.

Assigning `SECURITY_ZONE_LOCAL` as the default security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpDownloadFile, HttpGetDefaultZoneId, HttpGetFileZoneInfo, HttpSetFileZoneInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetEncodingType Function

```
INT WINAPI HttpSetEncodingType(
    HCLIENT hClient
    INT nEncodingType
);
```

The **HttpSetEncodingType** function specifies the type of encoding to be applied to the content of a HTTP request.

Parameters

hClient

Handle to the client session.

nEncodingType

Specifies the content encoding type for the request. The following values are defined:

Value	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request and no default content type will be specified.
HTTP_ENCODING_URL	Non-printable and extended ASCII characters will be encoded so they can be safely used with URLs and form data. Encoded characters will be represented by a percent symbol prefix, followed by a two digit hexadecimal value which represents the ASCII character code. This encoding is typically used with web services which process HTML form data. If the content type has not been explicitly specified, it will default to **application/x-www-form-urlencoded**.
HTTP_ENCODING_XML	This encoding specifies the content being submitted to the server contains XML structured data. If the content type has not been defined, it will default to **application/xml**, which is the official MIME media type for XML text. Legacy services may expect **text/xml** as the content type.
HTTP_ENCODING_JSON	This encoding specifies the content being submitted to the server contains JSON structured data. If the content type has not been defined, it will default to **application/json**, which is the official MIME media type for JSON text. In rare cases, legacy services may expect **text/x-json** as the content type.
HTTP_ENCODING_UTF8	This encoding specifies the content is unstructured text which has been UTF-8 encoded. If the content type has not been defined, it will default to **text/plain; charset=utf-8** which is appropriate for any kind of general text payloads. If you use the Unicode version of the functions in this library, all text is normally submitted as UTF-8 encoded text.
HTTP_ENCODING_ANSI	This encoding specifies the content is unstructured text and may not be UTF-8 encoded if the ANSI version of the functions in this library are used. This encoding is not

	recommended unless there is a specific need to send text using a specific character set. If the content type has not been defined, it will default to **`plain/text`**.
HTTP_ENCODING_XMLURL	This encoding is identical to URL encoding, except spaces are not encoded. It is used with legacy web services which expect form data in an XML format and cannot process encoded whitespace. This encoding should not be specified for services which use REST APIs. If the content type has not been specified, it will default to **`application/x-www-form-urlencoded`**.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetEncodingType** function sets the type of encoding used when optional parameter data is submitted with a request for a resource. If you do not set the Content-Type request header prior to submitting the request, the default content type provided to the server will be determined by the encoding you choose. If you specify HTTP_ENCODING_NONE, no default content type will be provided. Because most services require a Content-Type header as part of a POST request, this can cause the server to reject the request. The **HttpSetContentType** function can be used to explicitly specify the content type for the request.

If the service you are connected with expects an XML formatted request, it is recommended you use the **HttpPostXml** function. If the service expects JSON formatted text, use the **HttpPostJson** function. These functions ensure the correct encoding and content type are always used. If the service expects some other kind of text payload, you can set the appropriate content type using the **HttpSetContentType** function and then use the **HttpPostText** function to submit the request.

If you are using the **HttpPostData** function to submit form data to the server, in most cases you should use HTTP_ENCODING_URL. If the form data contains XML text, you may need to use the HTTP_ENCODING_XMLURL encoding type. However, in most cases, this is not the appropriate encoding type for XML and should never be used when submitting XML to a REST API endpoint.

If you are submitting a payload which contains binary data, you would typically use HTTP_ENCODING_NONE and explicitly set the appropriate content type using the **HttpSetContentType** function. For example, if you're submitting a JPEG image to a REST endpoint, you should not use any encoding and set the Content-Type header to **`image/jpeg`**. If your application needs to set the content type to a non-standard value for a proprietary API, use the **HttpSetRequestHeader** function to set the Content-Type header value. That function will not perform any validation on the header value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetEncodingType, HttpPostData, HttpPostJson, HttpPostText, HttpPostXml, HttpSetContentType,

HttpSetRequestHeader, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetFileZoneInfo Function

```
BOOL WINAPI HttpSetFileZoneInfo(
   LPCTSTR lpszLocalFile,
   DWORD dwZoneId,
   LPCTSTR lpszHostUrl
);
```

The **HttpSetFileZoneInfo** function sets or clears the zone identifier information for a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of a local file. This parameter must specify a regular file which exists on the local system and the current process must have permission to access and write to the file. If the file path includes environment variables surrounded in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL.

dwZoneId

An unsigned integer which specifies the zone ID which should be assigned to the file. It may be one of the following values:

Value	Description
SECURITY_ZONE_NONE	The file does not have a zone identifier associated with it.
SECURITY_ZONE_LOCAL	The file is considered to be part of the local computer zone.
SECURITY_ZONE_INTRANET	The file originates from the local intranet zone, such as an internal corporate network.
SECURITY_ZONE_TRUSTED	The file originates from a site in the **Trusted Sites** zone.
SECURITY_ZONE_INTERNET	The file was downloaded from the Internet and is treated with caution by Windows.
SECURITY_ZONE_RESTRICTED	The file originates from a site in the **Restricted Sites** zone, which is used for untrusted content.

lpszHostUrl

An optional pointer to a string which specifies the URL used to originally download the file. If this information is not required the parameter may be a NULL pointer or an empty string. If an originating URL is provided, it must be a valid HTTP URL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetFileZoneInfo** function enables applications to set or remove the security zone identifier metadata associated with a local file. This information is stored in an alternate data stream named `:Zone.Identifier` and is used by Windows to determine the origin and trust level of a file, potentially prompting the user with a warning before the file is opened.

The file name specified by the *lpszLocalFile* parameter must not reference a network location. This

includes UNC paths (such as `\\server\share\file.txt`) and drive letters that are mapped to network shares. Security zone metadata cannot be applied to files on network volumes because the SMB protocol does not support alternate data streams. If a file with zone metadata is copied from a local NTFS drive to a network share, any associated streams like `Zone.Identifier` will be lost.

Specifying `SECURITY_ZONE_NONE` will remove any existing zone identifier stream from the file. This has the same effect as selecting the Unblock option in the file's properties dialog in Windows Explorer. Once the stream is removed, the file will be treated as though it originated from the local computer. Note that when the `Zone.Identifier` stream is deleted from a file to remove its security zone information, Windows Explorer may not immediately reflect this change. This is because Explorer caches certain file metadata, including zone information used to display the "This file came from another computer" warning in the file's Properties dialog.

When assigning a zone ID, it is important to ensure that the value accurately reflects the source of the file. For files that have been downloaded from the Internet, it is strongly recommended to use `SECURITY_ZONE_INTERNET`. This allows Windows and other software to enforce appropriate security measures, such as requiring user confirmation before executing or opening the file.

In most scenarios, zone identifiers should only be set for files that were downloaded from untrusted or external sources. Setting an overly permissive zone (such as `SECURITY_ZONE_LOCAL`) for files obtained from the Internet or other potentially unsafe locations can expose users to security risks and circumvent built-in protections.

If a URL is provided via the *lpszHostUrl* parameter, it will be included in the zone metadata and can be retrieved later using the **HttpGetFileZoneInfo** function. This provides additional context about the origin of the file and may aid in auditing or diagnostic tasks. The URL must be a syntactically valid HTTP address and should reflect the actual location from which the file was downloaded.

Not all file systems support alternate data streams. The function will fail if used on a volume (such as FAT32) that does not support this feature. Additionally, modifying zone information may fail if the file is in use, read-only, or if the current process does not have the required permissions. It is possible to disable the creation of security zone metadata through local policy settings. In this case, the function will fail and the last error code will be set to `ST_ERROR_POLICY_NO_ZONEINFO`.

For more detailed information about the security zone identifiers, refer to the **HttpSetDefaultZoneId** function.

Assigning `SECURITY_ZONE_LOCAL` as the security zone identifier for downloaded files is strongly discouraged. This zone disables key Windows security features and can expose users to significant risk. This security zone should only be used with files that are generated locally by the application and are guaranteed to be safe. Never assign this zone to content obtained from the Internet or other untrusted sources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDownloadFile, HttpGetDefaultZoneId, HttpGetFileZoneInfo, HttpSetDefaultZoneId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetFormProperties Function

```
INT WINAPI HttpSetFormProperties(
   HFORM hForm,
   LPHTTPFORMPROPERTIES lpFormProp
);
```

The **HttpSetFormProperties** function updates the properties of the specified form.

Parameters

hForm

Handle to the virtual form.

lpFormProp

Points to a HTTPFORMPROPERTIES structure which specifies the new properties for the form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCreateForm, HttpGetFormProperties, HttpSubmitForm, HTTPFORMPROPERTIES

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetLastError Function

```
VOID WINAPI HttpSetLastError(
   DWORD dwErrorCode
);
```

The **HttpSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those functions which call **HttpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **HttpGetLastError** function. The use of **HttpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetErrorString, HttpGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetOption Function

```
INT WINAPI HttpSetOption(
   HCLIENT hClient,
   DWORD dwOption,
   BOOL bEnabled
);
```

The **HttpSetOption** function enables or disables the specified option.

Parameters

hClient

Handle to the client session.

dwOption

An unsigned integer which specifies one of the following options:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.

HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

bEnabled

An integer value which determines if the option should be enabled or disabled. A non-zero value specifies that the option should be enabled, while a zero value specifies that the option should be disabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetOption** function can only enable or disable the options listed above. Other options are only available when the connection is initially established. For example, you cannot use this function to enable security options or switch to using HTTP/2 after the connection has been made.

If you use HTTP_OPTION_FREETHREAD to permit any thread to reference a client handle allocated in another thread, your application is responsible for ensuring it does not attempt to submit requests using the same handle on different threads at the same time. If two different threads attempt to perform an operation using the same handle, there is no guarantee as to which thread will complete the operation first.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpConnect, HttpGetOption, HttpProxyConnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetPriority Function

```
INT WINAPI HttpSetPriority(
   HCLIENT hClient,
   INT nPriority
);
```

The **HttpSetPriority** function specifies the priority for file transfers.

Parameters

hClient

Handle to the client session.

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Value	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the function succeeds, the return value is the previous file transfer priority. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetPriority** function can be used to control the processor usage, memory and network

bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the HTTP_PRIORITY_BACKGROUND value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The HTTP_PRIORITY_CRITICAL priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application.

If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to HTTP_PRIORITY_CRITICAL, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetRequestHeader Function

```
BOOL WINAPI HttpSetRequestHeader(
    HCLIENT hClient,
    LPCTSTR lpszHeader,
    LPCTSTR lpszValue
);
```

The **HttpSetRequestHeader** function sets the value of the specified request header field.

Parameters

hClient

Handle to the client session.

lpszHeader

A pointer to a null-terminated string which specifies the name of the header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszValue

A pointer to a null-terminated string which specifies the value of the HTTP header. Headers values cannot contain carriage-return or linefeed characters. If the value contains Unicode characters, they will be UTF-8 encoded. If this parameter or NULL or a zero-length string, the header field is deleted from the request headers.

Return Value

If the function succeeds, the return value is non-zero. If the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetRequestHeader** function defines a header field and value that is submitted to the server when a resource is requested. If the header field has already been defined, it will be replaced by the new value. There are a number of header fields which are automatically created by the library, and others that are conditionally created depending on whether or not certain options are specified. The following header fields are automatically created by the library:

Header Field	Description
Accept	This header specifies the types of resources that are acceptable to the client. By default, all resource data types are accepted by the client.
Authorization	This header is automatically created if the client uses the HttpAuthenticate function to authenticate a client session.
Connection	This header determines if the connection is maintained after a resource has been requested, or if the connection should be immediately closed. The value of this header depends on whether the HTTP_OPTION_KEEPALIVE option has been specified.
Content-Length	This header defines the length of the data that is being posted to the server or the size of a file being uploaded to the server.
Content-Type	This header defines the content type for data posted to the server.

	This header is automatically created if required.
Host	This header specifies the name of the server that the client has connected to. This is automatically generated when any resource is requested.
Pragma	This header is used to control caching performed by the server. If the option HTTP_OPTION_NOCACHE has been specified, this header will automatically be defined with the value "no-cache".
Proxy-Authorization	This header is automatically created if a proxy connection has been established and a username and password is required to authenticate the client session.

Request headers are generated by functions that send resource requests. In some cases, header values are supplied by the requesting function only if the application has not previously defined the header. For others, the requesting function overrides what the application may have defined.

If you use this function to set the Authorization header to a custom value for this client session, you must not call the **HttpAuthenticate** function. The **HttpAuthenticate** function will always override any custom Authorization header value and replace it with the credentials token generated from the username and password provided. If your application was using this function to specify an OAuth 2.0 bearer token, it is recommended you use the **HttpSetBearerToken** function instead.

If you need to change the default content type for the next request, it is recommended you use the **HttpSetContentType** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticate, HttpClearHeaders, HttpGetFirstHeader, HttpGetNextHeader, HttpGetRequestHeader, HttpGetResponseHeader, HttpSetBearerToken, HttpSetContentType, HttpSetEncodingType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetTimeout Function

```
INT WINAPI HttpSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **HttpSetTimeout** function sets the number of seconds the client will wait for a response from the server.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete. If this value is zero, the default timeout period of 20 seconds will be used.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller. The timeout value affects both the requests submitted and the responses returned by the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetTimeout, HttpIsBlocking, HttpIsReadable, HttpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpStartQueue Function

```
BOOL WINAPI HttpStartQueue(
  HQUEUE hQueue,
  DWORD dwQueueMode,
  DWORD dwReserved,
  HTTPEVENTPROC lpEventProc,
  DWORD_PTR dwParam
);
```

The **HttpStartQueue** function begins transferring files in the queue.

Parameters

hQueue

A handle to a file transfer queue.

dwQueueMode

An unsigned integer which specifies which files in the queue should be transferred. It may be one of the following values:

Constant	Description
HTTP_QUEUE_ALL	All files in the queue should be transferred.
HTTP_QUEUE_DOWNLOAD	Transfer only those files which have been queued for download.
HTTP_QUEUE_UPLOAD	Transfer only those files which have been queued for upload.

dwReserved

An unsigned integer reserved for internal use. This parameter value should always be zero.

lpEventProc

Specifies the procedure-instance address of an application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, there will be no callback notifications.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

Queued file transfers are performed asynchronously using a background worker thread. If you provide the address to an event callback function, that function will always be invoked in the context of the queue manager thread. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself and you should avoid calling any functions which could cause the thread to block. For example, you should not attempt to establish other network

connections from within the event handler.

If your application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function. Controls should only be modified by the same UI thread that created their window. One common approach to address this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

To obtain the handle to the queue from within an event handler, call the **HttpGetThreadQueue** function with the ***dwThreadId*** parameter set to a value of zero. This will return the handle for the current queue management session which invoked the callback function.

If a callback function is provided, the HTTP_QUEUE_EVENT event notification will be invoked whenever the internal state of the queue has changed. The handle value passed to the callback function will be a handle to the queue, not to a client connection. The **HttpGetQueueStatus** function can be called within the event handler to obtain the current state of the queue. This will event will always occur when the queue manager begins processing files in the queue and after it has stopped. This event will also occur whenever the queue is suspended and then resumed.

The **HttpWaitForQueue** function can be used to wait for the queue manager to complete processing the transfer queue.

Example

```
// Create a new queue
HQUEUE hQueue = HttpCreateQueue(INFINITE, HTTP_TIMEOUT, HTTP_OPTION_DEFAULT);

if (hQueue == INVALID_QUEUE)
    return;

// Add a file to the queue
DWORD dwFileId = HttpAddQueuedFile(
                     hQueue,
                     lpszLocalFile,
                     lpszRemoteFile,
                     HTTP_OPTION_DEFAULT,
                     HTTP_TIMEOUT);

// Display information about the queued file
if (dwFileId != 0)
{
    HTTPQUEUEDFILE queuedFile;

    if (HttpGetQueuedFile(hQueue, dwFileId, &queuedFile))
    {
        switch (queuedFile.dwMode)
        {
        case HTTP_QUEUE_DOWNLOAD:
               _tprintf(_T("%u: Download \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szRemoteFile,
                        queuedFile.szLocalFile);
                break;

        case HTTP_QUEUE_UPLOAD:
               _tprintf(_T("%u: Upload \"%s\" to \"%s\"\n"),
                        dwFileId,
                        queuedFile.szLocalFile,
```

```
                        queuedFile.szRemoteFile);
                break;
            }
        }
    }

    // Start the queued file transfers and wait for it to complete
    if (HttpStartQueue(hQueue, HTTP_QUEUE_ALL, 0, NULL, 0))
    {
        HttpWaitForQueue(hQueue, INFINITE, &dwElapsed, &dwError);
        HttpStopQueue(hQueue);
    }

    // Remove all files from the queue
    HttpDeleteQueue(hQueue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpEventProc, HttpGetQueueStatus, HttpGetThreadQueue, HttpResumeQueue, HttpStartQueue, HttpStopQueue, HttpSuspendQueue, HttpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpStopQueue Function

```
BOOL WINAPI HttpStopQueue(
    HQUEUE hQueue
);
```

The **HttpStopQueue** function stops transferring queued files.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

When the **HttpStopQueue** function is called and there is a file transfer in progress, it will not immediately stop the upload or download of the file. Instead, the queue manager is signaled to stop processing additional files in the queue after the transfer has completed. To wait for the current transfer to complete, call the **HttpWaitForQueue** function.

It is permitted to call **HttpStopQueue** from within a queue event handler. In this case, the current file transfer will complete and the queue manager will terminate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCancelQueuedFile, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpSuspendQueue, HttpWaitForQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSubmitForm Function

```
INT WINAPI HttpSubmitForm(
   HCLIENT hClient,
   HFORM hForm,
   LPVOID lpvResult,
   LPDWORD lpcbResult,
   DWORD dwOptions
);
```

The **HttpSubmitForm** function submits the contents of the specified form to a script on the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

hForm

Handle to the virtual form which contains the data to be submitted to the server.

lpvResult

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvResult*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_SUBMIT_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_SUBMIT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSubmitForm** function is used to submit form data to a script that executes on the server and then copy the output from that script into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the ***lpvResult*** parameter will point to the buffer that was allocated by the client and the value that the ***lpcbResult*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvResult*** parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the ***lpcbResult*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HFORM hForm = INVALID_FORM;
HGLOBAL hgblResult = (HGLOBAL)NULL;
DWORD cbResult = 0;
INT nResult = 0;

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);

if (hForm == INVALID_FORM)
    return;

HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nResult = HttpSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);
HttpDestroyForm(hForm);

if (hgblResult != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer points to data returned by the server after the form
    // data was submitted

    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddFormField, HttpAddFormFile, HttpClearForm, HttpCreateForm, HttpDeleteFormField, HttpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSubmitRequest Function

```
INT WINAPI HttpSubmitRequest(
   DWORD dwRequestType,
   LPCTSTR lpszResource,
   LPHTTPCONNECTION lpConnection,
   LPCVOID lpvRequest,
   DWORD dwRequestLength,
   LPVOID lpvResponse,
   LPDWORD lpdwResponseLength
);
```

The **HttpSubmitRequest** function provides a high-level interface for submitting requests to a web service.

Parameters

dwRequestType

An unsigned integer which specifies the type of request being submitted to the server. It may be one of the following values:

Value	Description
HTTP_METHOD_DEFAULT	Perform a default request action based on the value of parameters passed to this function. If there is no request payload, the action will default to HTTP_METHOD_GET, otherwise it will default to HTTP_METHOD_POST. It is generally recommended that you explicitly specify the request method; however, this default value can be useful if your application does not know if a payload will be included with the request at the time the function is called.
HTTP_METHOD_GET	Use the GET method when submitting the request to the server. If this action is used, the ***lpvRequest*** and ***dwRequestLength*** parameters will be ignored. The buffer specified by the ***lpvResponse*** parameter will contain the server's response to the request.
HTTP_METHOD_POST	Use the POST method when submitting the request to the server. The payload specified by the ***lpvRequest*** parameter will be included with the request and the buffer specified by the ***lpvResponse*** parameter will contain the server's response to the request.
HTTP_METHOD_PUT	Use the PUT method when submitting the request to the server. The payload specified by the ***lpvRequest*** parameter will be included with the request. This method is typically used with persistent storage on the server, such as uploading the contents of a file. If the ***lpvResponse*** parameter is not NULL, it will contain any response from the server after the operation has completed.

lpszResource

A null-terminated string which specifies the complete URL for the resource. Both standard and

secure connections are supported, and query parameters may be included with the URL. This parameter cannot be NULL and must specify an absolute (fully qualified) URI which begins with either http:// or https://.

lpConnection

A pointer to a **HTTPCONNECTION** structure which contains additional information about the client connection. This structure is required if your application must connect through a proxy server or when authentication is required to access a resource. This parameter can be NULL, in which case default values will be used when establishing the connection.

lpvRequest

A buffer which contains the payload data submitted to the server as part of the request. This parameter may point to a byte buffer or a null-terminated string. If there is no payload required for the request this parameter may be NULL. If the Unicode version of this function is called and the request buffer contains text, it will be automatically converted to UTF-8 prior to submitting the data to the server. If the ***dwRequestLength*** parameter is -1, the payload will always be processed as a null-terminated string and the function will fail if the buffer does not contain text characters.

dwRequestLength

An unsigned integer which specifies the size of the request payload. If the payload contains text, this should specify the number of characters in the buffer. If the payload contains binary data, this value should specify the number of bytes in the buffer. If the ***lpvRequest*** parameter points to a null-terminated string, this value can be -1 and the length of the payload will be determined automatically. If the Unicode version of this function is called and the length is -1, the request payload will be UTF-8 encoded prior to being submitted to the server.

lpvResponse

A buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns. This parameter may be NULL if the response data is not required. If this parameter is NULL, the ***lpdwResponseLength*** parameter must also be NULL. If a response buffer is provided, the data will always be returned as a byte stream, even when the Unicode version of this function is called. If the server returns a text payload, it will typically be UTF-8 encoded and the application can convert the text to Unicode using the **MultiByteToWideChar** function.

lpdwResponseLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvResponse*** parameter. If the ***lpvResponse*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned. If the ***lpvResponse*** parameter is not NULL, this parameter cannot be NULL.

Return Value

If the function succeeds, the return value is the server status code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**. The application should always check the return value to determine if the request was successful. If the server returns a 4xx or 5xx status code, this indicates the request was not accepted and the response buffer will typically contain information about the cause of the failure.

Remarks

The **HttpSubmitRequest** function provides a single high-level interface for submitting requests to a web server and returning the server's response. This function is entirely self-contained and does not require a client session handle. The current thread will block until the function returns and will not

generate event notifications. If you need to download a large file from a web server and want progress updates, it is recommended you use the **HttpDownloadFile** function.

If the service you are using requires authentication, you will need to allocate a **HTTPCONNECTION** structure and set the appropriate member values to specify the authentication method and user credentials. This structure also allows you to provide additional information, such as additional request header values, a custom timeout period and additional connection options.

When the *lpvRequest* parameter is not NULL, the contents of the buffer will be examined to determine if it contains text or binary data. If the *dwRequestLength* parameter is -1, the *lpvRequest* parameter is always considered to be a pointer a null-terminated string, with the length calculated by counting the number of characters up to the terminating null character. If the request payload is text, the function will also attempt to determine if the payload looks like XML or JSON and will automatically set the appropriate content type. The application can explicitly provide a MIME content type by setting the *lpszContentType* member of the **HTTPCONNECTION** structure.

Example

```
// Submit an XML payload to the server and return the response in a global
// memory buffer allocated by the function
LPCTSTR lpszResource = _T("https://www.example.com/postxml");
LPCTSTR lpszRequest = _T("<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n") \
                      _T("<test>\r\n") \
                      _T("  <data param=\"A\">Value1</data>\r\n") \
                      _T("  <data param=\"B\">Value2</data>\r\n") \
                      _T("</test>\r\n");

// Provide additional connection information which includes user credentials
// and a custom header which includes an API token for the service; note that
// we need to initialize the dwSize member of the structure
HTTPCONNECTION httpConnection = { 0, };
httpConnection.dwSize = sizeof(httpConnection);
httpConnection.nAuthType = HTTP_AUTH_BASIC;
httpConnection.lpszUserName = _T("userid");
httpConnection.lpszPassword = _T("secret");
httpConnection.lpszContentType = _T("application/xml");
httpConnection.lpszHeaders = _T("X-API-Token: 99d2fe39-0246-4efa-98e0-
4d775579fa5d");

// Declare the global memory handle which will contain the server response
// and initialize the length to 0, which tells the API that a HGLOBAL handle
// is being used rather than a pointer to a pre-allocated buffer
HGLOBAL hgblResponse = NULL;
DWORD dwLength = 0;

INT nResult = HttpSubmitRequest(
    HTTP_METHOD_POST,
    lpszResource,
    &httpConnection,
    lpszRequest,
    (DWORD)-1,
    &hgblResponse,
    &dwLength);

if (nResult != HTTP_ERROR && hgblResponse != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResponse);
```

```
    // The lpBuffer variable points to the data returned by the server and
    // the application is responsible for freeing the global memory handle

    GlobalUnlock(hgblResponse);
    GlobalFree(hgblResponse);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDownloadFile, HttpGetData, HttpPostData, HttpPostJson, HttpPostXml, HttpUploadFile, HTTPCONNECTION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSuspendQueue Function

```
BOOL WINAPI HttpSuspendQueue(
    HQUEUE hQueue
);
```

The **HttpSuspendQueue** function pauses all file transfers for the specified queue.

Parameters

hQueue

A handle to a file transfer queue.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

If there is a file transfer in progress when this function is called, the transfer will complete normally. The queue manager will enter a suspended state after the transfer has completed and before it begins processing the next file in the queue.

It is permitted to call **HttpSuspendQueue** from within a queue event handler. In this case, the current file transfer will complete and the queue manager will stop processing additional files in the queue until it is resumed or stopped.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpCancelQueuedFile, HttpGetQueueStatus, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpStopQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpTaskAbort Function

```
BOOL WINAPI HttpTaskAbort(
   UINT nTaskId,
   DWORD dwMilliseconds
);
```

Abort the specified asynchronous task.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to abort.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskAbort** function signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the *dwMilliseconds* parameter has a value of zero, the function returns immediately after the background thread has been signaled. If the *dwMilliseconds* parameter is non-zero, the function will wait that amount of time for the background thread to terminate.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock. To abort a file transfer within an event handler, use the **HttpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskDone, HttpTaskResume, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpTaskDone Function

```
BOOL WINAPI HttpTaskDone(
    UINT nTaskId
);
```

Determine if an asynchronous task has completed.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed, this function returns a non-zero value. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskDone** function is used to determine if the specified asynchronous task has completed. If you use this function to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this function rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **HttpGetTaskError** function will return the last error code associated with the task. A return value of zero indicates success, while a non-zero return value specifies an error code that indicates the cause of the failure. The last error code for the task can also be retrieved using the **HttpTaskWait** function, which causes the application to wait for the asynchronous task to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetTaskError, HttpTaskAbort, HttpTaskResume, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpTaskResume Function

```
BOOL WINAPI HttpTaskResume(
    UINT nTaskId
);
```

Resume execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskResume** function resumes execution of the background worker thread that was previously suspended using the **HttpTaskSuspend** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskAbort, HttpTaskDone, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpTaskSuspend Function

```
BOOL WINAPI HttpTaskSuspend(
    UINT nTaskId
);
```

Suspend execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskSuspend** function will suspend execution of the background worker thread associated with the task. Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **HttpTaskResume** function. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskAbort, HttpTaskDone, HttpTaskResume, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpTaskWait Function

```
BOOL WINAPI HttpTaskWait(
   UINT nTaskId,
   DWORD dwMilliseconds,
   DWORD dwReserved,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);
```

Wait for an asynchronous task to complete.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to complete.

dwReserved

An unsigned integer reserved for future use. This value should always be zero.

lpdwElapsed

A pointer to an unsigned integer that will contain elapsed time in milliseconds when the function returns. If this information is not required, this parameter may be NULL.

lpdwError

A pointer to an unsigned integer that will contain the error code associated with the completed task. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskWait** function waits for the specified task to complete. If the task is active and the ***dwMilliseconds*** parameter is non-zero, this function will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the ***dwMilliseconds*** parameter is zero, then this function will poll the status of the task and return immediately to the caller.

If the specified task has already completed at the time this function is called, the function will return immediately without causing the current thread to block. If the ***lpdwElapsed*** parameter is not NULL, it will contain the number of milliseconds that it took for the task to complete. If the ***lpdwError*** parameter is not NULL, it will contain the last error code value that was set by the worker thread before it terminated. If this value is zero, that means that the background operation was successful and no error occurred. A non-zero value will indicate that the background operation has failed.

You should not call this function from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this function is blocked waiting for the background task to complete, and this can cause your application to appear non-

responsive to the end-user. If you have a GUI application and you need to periodically check to see if a task has completed, create a timer to periodically call the **HttpTaskDone** function. When it returns a non-zero value (indicating that the task has completed), you can safely call **HttpTaskWait** to obtain the elapsed time and last error code without blocking the current thread.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Example

```
UINT nTaskId;

// Begin a file transfer in the background

nTaskId = HttpAsyncGetFile(hClient,
                           lpszLocalFile,
                           lpszRemoteFile,
                           HTTP_TRANSFER_DEFAULT,
                           0,
                           NULL,
                           0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    HttpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpTaskDone, HttpTaskResume, HttpTaskSuspend, HttpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpUninitialize Function

```
VOID WINAPI HttpUninitialize();
```

The **HttpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **HttpInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **HttpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **HttpUninitialize** for every successful call to **HttpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpDisconnect, HttpInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpUploadFile Function

```
BOOL WINAPI HttpUploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPHTTPTRANSFERSTATUS lpStatus
   HTTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **HttpUploadFile** function uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.

HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpUploadFile** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then upload the file to the server. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

Not all web servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the URL specify a username and password to upload a file.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the method will fail and return a value of zero.

The ***lpStatus*** parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The ***lpEventProc*** parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The ***dwParam*** parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the ***this*** pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for ***lpEventProc*** and the ***dwParam*** parameter will be ignored.

The **HttpUploadFile** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpPutFile** to upload the file. If the server does not support the PUT command, you may be able to upload files using the **HttpPostFile** function. Refer to that function for more information.

Example

```
HTTPTRANSFERSTATUS httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile =
_T("https://update:secret@www.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = HttpUploadFile(lpszLocalFile,
                         lpszRemoteFile,
                         HTTP_OPTION_DEFAULT,
                         &httpStatus,
                         NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDownloadFile, HttpEventProc, HttpGetFileZoneInfo, HttpGetTransferStatus, HttpPostFile, HttpPutFile, HttpSetFileZoneInfo, HttpSubmitRequest, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpUploadFileEx Function

```
BOOL WINAPI HttpUploadFileEx(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   UINT nTimeout
   DWORD dwOptions
   LPHTTPTRANSFERSTATUSEX lpStatus
   HTTPEVENTPROC lpEventProc
   DWORD_PTR dwParam
);
```

The **HttpUploadFileEx** function uploads the specified file from the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

lpszLocalFile

A null-terminated string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A null-terminated string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration

	can be changed using the system Control Panel.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times

	for smaller amounts of data over fast network connections.

lpStatus

A pointer to an HTTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpUploadFileEx** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then upload the file to the server. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] hostname [:port] / [path / ...]
[filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

Not all web servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the URL specify a username and password to upload a file.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the method will fail and return a value of zero.

The ***lpStatus*** parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The ***lpEventProc*** parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The ***dwParam*** parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the ***this*** pointer as the value, and then cast it back to an object pointer inside the

callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpUploadFileEx** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpPutFileEx** to upload the file. If the server does not support the PUT command, you may be able to upload files using the **HttpPostFile** function. Refer to that function for more information.

Example

```
HTTPTRANSFERSTATUSEX httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszRemoteFile =
_T("http://update:secret@www.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = HttpUploadFileEx(lpszLocalFile,
                           lpszRemoteFile,
                           HTTP_OPTION_DEFAULT,
                           &httpStatus,
                           NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEventProc, HttpDownloadFileEx, HttpGetTransferStatusEx, HttpPostFile, HttpPutFileEx, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpValidateHostName Function

```
BOOL WINAPI HttpValidateHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **HttpValidateHostName** function determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The function will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the ***nMaxLength*** parameter is zero, it will be ignored and the IP address will not be returned.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the ***lpszAddress*** string buffer. The buffer must be large enough to store the complete address. Because this function can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the ***lpszAddress*** parameter will be ignored.

Return Value

If the function succeeds, the host name is valid and the return value will be non-zero. If the function fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpValidateHostName** function provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. If the Unicode version of this function is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

If the ***lpszHostName*** parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address.

If you wish to validate a complete HTTP URL instead of a host name, use the **HttpValidateUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpValidateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpValidateUrl Function

```
BOOL WINAPI HttpValidateUrl(
    LPCTSTR lpszUrl
);
```

The **HttpValidateUrl** function determines if a string represents a valid HTTP URL.

Parameters

lpszUrl

A null-terminated string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpValidateUrl** function will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure HTTP scheme. This function will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this function will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **HttpConnectUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpConnectUrl, HttpValidateHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpVerifyFile Function

```
BOOL WINAPI HttpVerifyFile(
   HCLIENT hClient,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszRemoteFile,
   DWORD dwReserved
);
```

The **HttpVerifyFile** function attempts to verify that the size of a file on the local system is the same as the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A null-terminated string that specifies the name of file on the local system.

lpszRemoteFile

A null-terminated string that specifies the name of the file on the server.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpVerifyFile** function will attempt to verify that the contents of the local and remote files are identical by comparing the size of the files. This function is provided for compatibility with the File Transfer Protocol API, and should not be considered a reliable method for comparing files. Web servers may not consistently return file size information for dynamically created content such as HTML pages which use server-side includes.

It is not recommended that you use this function with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **HttpVerifyFile** function to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDeleteFile, HttpGetFile, HttpGetFileSize, HttpPutFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpWaitForQueue Function

```
BOOL WINAPI HttpWaitForQueue(
   HQUEUE hQueue,
   DWORD dwMilliseconds,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);
```

The **HttpWaitForQueue** function waits for the specified queue to complete the file transfers.

Parameters

hQueue

A handle to a file transfer queue.

dwMilliseconds

An unsigned integer value which specifies the number of milliseconds to wait for the queue to complete processing. If this value is zero, the function will return immediately. If this value is INFINITE (0xFFFFFFFF), the function will block indefinitely until all files in the queue have been processed.

lpdwElapsed

A pointer to an unsigned integer which will contain the number of milliseconds the queue manager has been active when the function returns. If this information is not needed, this parameter can be NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

lpdwError

A pointer to an unsigned integer which will contain the last error code set if a file transfer has failed or was canceled. If this information is not needed, this parameter can be NULL. If this parameter is not NULL, it will always be initialized to a value of zero.

Return Value

If the function succeeds, the return value is non-zero, which means the queue manager has completed transferring all queued files. If the function fails, or the timeout period elapses, it will return zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function will cause the current thread to block until either the queue manager has completed processing all files in the queue, or until the specified number of milliseconds have elapsed. It is important to note that Windows messages will not be processed during this time. If you call this function within the main UI thread, it can potentially cause the application to become non-responsive. To determine the current state of the queue without blocking the current thread, call the **HttpGetQueueStatus** function.

If the *dwMilliseconds* parameter is non-zero and file transfers have not completed within the specified amount of time, the function will return zero and the last error code will be set to `ST_ERROR_OPERATION_TIMEOUT`.

If an event callback function has been specified when starting the queue, you should never call this function within the event handler. Attempting to do so can potentially create a situation which will cause the application to become non-responsive. The function will return zero if it determines it's being called within the context of the queue manager thread and will set the last error code to `ST_ERROR_THREAD_DEADLOCK`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetQueueStatus, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpStopQueue, HttpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpWrite Function

```
INT WINAPI HttpWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **HttpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for an asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpIsBlocking, HttpIsReadable, HttpIsWritable, HttpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Data Structures

- HTTPCONNECTION
- HTTPFORMPROPERTIES
- HTTPQUEUEDFILE
- HTTPQUEUESTATUS
- HTTPTRANSFERSTATUS
- HTTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPCONNECTION

The **HTTPCONNECTION** structure provides additional information used when submitting a request to the server.

```
typedef struct _HTTPCONNECTION
{
    DWORD   dwSize;
    DWORD   dwOptions;
    DWORD   dwVersion;
    UINT    nTimeout;
    UINT    nEncoding;
    UINT    nProxyType;
    UINT    nProxyPort;
    UINT    nAuthType;
    LPCTSTR lpszProxyServer;
    LPCTSTR lpszProxyUser;
    LPCTSTR lpszProxyPassword;
    LPCTSTR lpszUserAgent;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
    LPCTSTR lpszContentType;
    LPCTSTR lpszHeaders;
    LPSECURITYCREDENTIALS lpCredentials;
} HTTPCONNECTION, *LPHTTPCONNECTION;
```

Members

dwSize

The size of this structure in bytes. If this member is not initialized to the correct value, the structure will be considered invalid.

dwOptions

A bitmask of supported options. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control

	panel.
HTTP_OPTION_NOUSERAGENT	This option specifies that the client should not include a User-Agent header with requests made during the session. The User-Agent header is typically used by servers to identify the client application, operating system and other platform information. This option may be used when the server does not require a User-Agent header.
HTTP_OPTION_HTTP2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. The value of the ***dwVersion*** member will be ignored when this option is used.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects secure connections.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server even if the request URL does not use the **https://** scheme. If this option is not specified, the type of connection depends on the URL scheme provided when submitting the request.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.

dwVersion

A value which specifies the requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. If this value is zero, the default protocol version will be used. The HTTPVERSION macro can be used to create a version value, or the following values can be used:

Value	Description
HTTP_VERSION_DEFAULT	The client should use the default protocol version for this

	session. This is the same as specifying HTTP_VERSION_11 and is recommended to ensure the broadest compatibility with most servers.
HTTP_VERSION_10	The client should use the HTTP/1.0 protocol standard originally defined in RFC 1945. This version of the protocol supports the use of request header blocks, however it does not support persistent connections or chunked data and has limited cache control mechanisms.
HTTP_VERSION_11	The client should use the HTTP/1.1 protocol standard defined in RFC 2616 and RFC 7230. This is the most widely used version of the protocol and is the default for client connections. It is recommended most applications use this version.
HTTP_VERSION_20	The client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. If the client or server does not support HTTP/2, the client will automatically attempt to use an earlier version of the protocol.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the operation. If this value is zero, a default timeout period of 20 seconds will be used.

nEncoding

A value which specifies the content encoding type for the request. If the value is zero, no encoding will be used. The following values are defined:

Value	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request and no default content type will be specified. This encoding type should be used with REST APIs and other services which expect XML or JSON request payloads.
HTTP_ENCODING_URL	Non-printable and extended ASCII characters will be encoded so they can be safely used with URLs and form data. Encoded characters will be represented by a percent symbol prefix, followed by a two digit hexadecimal value which represents the ASCII character code. This encoding is typically used with web services which process HTML form data.
HTTP_ENCODING_XML	This encoding is identical to URL encoding, except spaces are not encoded. It is used with legacy web services which expect form data in an XML format and cannot process encoded whitespace. This encoding should not be specified for services which use REST APIs.

nProxyType

A value which specifies the type of proxy server the client is connecting to. A value of zero specifies no proxy server is being used and the client should establish a direct connection to the server. The following proxy server types are supported:

Value	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the ***dwOptions*** parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server settings configured in Windows.

nProxyPort

A value which specifies the port the proxy server will use to accept connections. This value is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

nAuthType

A value which specifies the type of authentication used to establish the connection. If the value is zero, the client session will not be authenticated. The following values are defined:

Value	Description
HTTP_AUTH_NONE	No client authentication should be performed. The ***lpszUserName*** and ***lpszPassword*** members are ignored.
HTTP_AUTH_BASIC	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the ***lpszPassword*** member must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error.

lpszProxyServer

A null-terminated string which identifies the proxy server which should be used to establish the connection. If this value is ignored if the ***nProxyType*** member is zero. If a proxy server type has been been specified, this value cannot be a NULL pointer or an empty string.

lpszProxyUser

A null-terminated string which specifies the user name used to authenticate the connection to the proxy server. If the proxy server does not require user authentication, this member can be a NULL pointer or an empty string. This member is ignored if the ***nProxyType*** member is zero.

lpszProxyPassword

A null-terminated string which specifies the password used to authenticate the connection to the proxy server. If the proxy server does not require user authentication, this member can be a NULL pointer or an empty string. This member is ignored if the ***nProxyType*** member is zero.

lpszUserAgent

A null-terminated string which identifies the application which is making a request to the server. If this value is NULL or an empty string, a default user agent string will be included with the request. To prevent a user agent from being included with the request, use the HTTP_OPTION_NOUSERAGENT option.

lpszUserName

A null-terminated string which specifies the user name used to authenticate the client session. This member may be NULL or an empty string if a user name is not required for the specified authentication type.

lpszPassword

A null-terminated string which specifies the password used to authenticate the client session. This member may be NULL or an empty string if a password is not required for the specified authentication type. If the authentication type is HTTP_AUTH_BEARER, this value must be the bearer token issued by the service provider.

lpszContentType

A null-terminated string which specifies the MIME content type for the request. If this member is NULL or specifies an empty string, the content type will be automatically determined based on the contents of the request payload. This value will only be used with PUT and POST methods.

lpszHeaders

A null-terminated string which specifies one or more additional header values which should be included with the request. The string must contain header name and value pairs separated by a colon, and multiple header values must be terminated with a carriage return/linefeed (CRLF) sequence after each value. The header values defined in this string will override any default values.

lpCredentials

A pointer to a **SECURITYCREDENTIALS** structure. This value is only used if the HTTP_OPTION_SECURE option is specified in the ***dwOptions*** member. This member may be NULL, in which case no client credentials will be provided to the server.

Remarks

This structure is used with the **HttpSubmitRequest** function to provide additional information used to establish a connection with the server. The ***dwSize*** member must be initialized to the actual size of the structure in bytes, and all other members should be initialized to a value of zero or NULL. If the structure is not initialized correctly, the function will fail and the last error code will be set to ST_ERROR_INVALID_PARAMETER.

Member values which are zero or NULL pointers will either be ignored or cause the function to use appropriate default values.

Applications should use HTTP_VERSION_DEFAULT as the value for the ***dwVersion*** parameter. This will default to an appropriate version for the Windows platform and ensures the broadest compatibility with most servers. If your application specifies HTTP_VERSION_20, a secure connection using TLS 1.2

or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

If your application requests a HTTP/2 connection and the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1. If a connection using an earlier version of the protocol cannot be established, the connection attempt will fail.

If the server requires a secure connection with a client certificate, the **HttpCreateSecurityCredentials** function can be used to allocate a credentials structure which can be assigned to the ***lpCredentials*** member. Call **HttpDeleteSecurityCredentials** after the **HttpSubmitRequest** function has returned to ensure the memory allocated for the security credentials has been freed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCreateSecurityCredentials, HttpSubmitRequest, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPFORMPROPERTIES

This structure is used by the **HttpGetFormProperties** and **HttpSetFormProperties** function to return and modify the properties of the specified form.

```
typedef struct _HTTPFORMPROPERTIES
{
   UINT    nFormMethod;
   UINT    nFormType;
   LPCTSTR lpszFormAction;
   DWORD   dwReserved1;
   DWORD   dwReserved2;
} HTTPFORMPROPERTIES, *LPHTTPFORMPROPERTIES;
```

Members

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. It may be one of the following values:

Value	Description
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. It may be one of the following values:

Value	Description
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

lpszFormAction

A null-terminated string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

dwReserved1

A reserved structure member.

dwReserved2

A reserved structure member.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCreateForm, HttpGetFormProperties, HttpSetFormProperties, HttpSubmitForm

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPQUEUEDFILE

This structure is used by the **HttpGetFirstQueuedFile**, **HttpGetNextQueuedFile** and **HttpGetQueuedFile** functions to return information about a file in the transfer queue.

```
typedef struct _HTTPQUEUEDFILE
{
    DWORD          dwFileId;
    DWORD          dwQueueMode;
    DWORD          dwQueueFlags;
    DWORD          dwTimeElapsed;
    DWORD          dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szLocalFile[HTTP_MAXFILENAMELEN];
    TCHAR          szRemoteFile[HTTP_MAXURLPATHLEN];
} HTTPQUEUEDFILE, *LPHTTPQUEUEDFILE;
```

Members

dwFileId

An unsigned integer which specifies a unique identifier for the queued file. The application should not make any assumptions about the value of the file identifiers. They should be considered opaque values which are only guaranteed to uniquely identify a file in the transfer queue. In particular, there is no guarantee that the file identifiers will be sequential and they should not be used as index values into an array.

dwQueueMode

An unsigned integer value which specifies how the file was queued for transfer. It may be one of the following values:

Constant	Description
HTTP_QUEUE_DOWNLOAD	The file was queued for download.
HTTP_QUEUE_UPLOAD	The file was queued for upload.

dwQueueFlags

An unsigned integer which specifies one or more bit flags which provides information about the status of the file transfer. It may be one or more of the following values:

Constant	Description
HTTP_QUEUE_FLAG_NONE	The file is pending transfer in queue.
HTTP_QUEUE_FLAG_COMPLETED	The file has been transferred successfully. If this flag is set, no errors were encountered during the upload or download.
HTTP_QUEUE_FLAG_FAILED	The file transfer failed. If this flag is set, the *dwError* member of this structure will contain the error code associated with the failed transfer.
HTTP_QUEUE_FLAG_CANCELED	The file transfer was canceled. This flag is only set when the **HttpCancelQueuedFile** function has been called and a queued file is in the process of being uploaded or downloaded.

dwTimeElapsed

An unsigned integer which specifies the number of milliseconds required to complete the transfer. This value will be zero unless the file has been transferred successfully.

dwLastError

An unsigned integer which specifies the last error code for a failed transfer. If the file was transferred successfully, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the file transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

szLocalFile

A null-terminated string which specifies the full path to the local file being transferred.

szRemoteFile

A null-terminated string which specifies the complete URL to the remote file being transferred.

Remarks

It is possible that the ***szLocalFile*** and ***szRemoteFile*** structure members will be different than the values passed to the **HttpAddQueuedFile** function. Those values are normalized, with any relative paths converted to absolute paths. Internationalized domain names will be encoded and the URL paths will be collapsed, removing any extraneous path information. For example, if the remote file name is specified as `https://www.server.tld/folder1/../folder2/filename.txt` it would be normalized as `https://www.server.tld/folder2/filename.txt`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddQueuedFile, HttpAddQueuedFileEx, HttpGetFirstQueuedFile, HttpGetNextQueuedFile, HttpGetQueuedFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPQUEUESTATUS

This structure is used by the **HttpGetQueueStatus** function to return information about the current status of the file transfer queue.

```
typedef struct _HTTPQUEUESTATUS
{
    DWORD dwStatus;
    DWORD dwThreadId;
    DWORD dwQueuedFiles;
    DWORD dwPendingFiles;
    DWORD dwCopiedFiles;
    DWORD dwFailedFiles;
    DWORD dwTimeElapsed;
    DWORD dwCurrentFile;
    DWORD dwLastError;
    ULARGE_INTEGER uiBytesCopied;
    ULARGE_INTEGER uiBytesTotal;
} HTTPQUEUESTATUS, *LPHTTPQUEUESTATUS;
```

Members

dwStatus

An unsigned integer which specifies the current status of the queue. It can be one of the following values:

Value	Description
HTTP_QUEUE_STATUS_IDLE	The file transfer queue is idle and no files are being uploaded or downloaded. This state is returned before **HttpStartQueue** has been called or after **HttpStopQueue** has been called. The queue will also automatically enter an idle state after the last file transfer has completed and the queue manager thread exits.
HTTP_QUEUE_STATUS_ACTIVE	Files in the queue are being uploaded or downloaded. The *dwCurrentFile* member of this structure identifies the file which is currently being transferred.
HTTP_QUEUE_STATUS_PAUSED	File transfers are currently paused. The queue enters this state after the **HttpSuspendQueue** function is called and resumes file transfers after the **HttpResumeQueue** function is called.

dwThreadId

An unsigned integer which specifies the thread ID for the queue manager which is performing the file transfers. All file transfers are performed asynchronously in a background worker thread. If this structure member is zero, the queue is idle and not performing file transfers. The application should not use the thread ID to obtain a handle to suspend or terminate the thread. This can potentially result in unexpected behavior or instability within the application.

dwQueuedFiles

An unsigned integer which specifies the total number of queued files. This value includes pending and completed file transfers.

dwPendingFiles

An unsigned integer which specifies the number of files which are queued to be transferred.

dwCopiedFiles

An unsigned integer which specifies the number of files which have been successfully transferred. This value reflects the total number of files which have the HTTP_QUEUE_FLAG_COMPLETED status.

dwFailedFiles

An unsigned integer which specifies the number of files which have failed to transfer or the file transfer has been canceled. This value reflects the total number of files which have the HTTP_QUEUE_FLAG_FAILED status.

dwTimeElapsed

An unsigned integer which specifies the total amount of time, in milliseconds, the queue manager has been active performing a file transfer. If the queue is idle, this value will reflect the total run time for the previously active queue. A value of zero indicates the queue was never active or the queue state has been reset with a call to **HttpResetQueue**.

dwCurrentFile

An unsigned integer which specifies the unique identifier for the current file being transferred. If no file transfer is in progress, this member will have a value of zero. If the value is non-zero, it can be passed to the **HttpGetQueuedFile** function to obtain information about the queued file.

dwLastError

An unsigned integer which specifies the last error code for a failed file transfer. If there have been no errors, this value will be zero.

uiBytesCopied

A value which specifies the number of bytes which were copied during the current or last transfer. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

uiBytesTotal

A value which specifies the total number of bytes which were copied during the current queue run. The ULARGE_INTEGER type is a union of a structure and a 64-bit value which can be used on both 32-bit and 64-bit systems. For languages other than C/C++ you can define this structure member as an unsigned 64-bit integer type.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtpv12.lib

See Also

HttpGetQueuedFile, HttpGetQueueStatus, HttpCancelQueuedFile, HttpResetQueue, HttpResumeQueue, HttpStartQueue, HttpStopQueue, HttpSuspendQueue

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPTRANSFERSTATUS

This structure is used by the **HttpGetTransferStatus** function to return information about a data transfer in progress.

```
typedef struct _HTTPTRANSFERSTATUS
{
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} HTTPTRANSFERSTATUS, *LPHTTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

HttpEnableEvents, HttpGetTransferStatus, HttpGetTransferStatusEx, HttpRegisterEvent, HTTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPTRANSFERSTATUSEX

This structure is used by the **HttpGetTransferStatusEx** function to return information about a data transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _HTTPTRANSFERSTATUSEX
{
   ULARGE_INTEGER uiBytesTotal;
   ULARGE_INTEGER uiBytesCopied;
   DWORD          dwBytesPerSecond;
   DWORD          dwTimeElapsed;
   DWORD          dwTimeEstimated;
} HTTPTRANSFERSTATUSEX, *LPHTTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values returned in the ***dwTimeElapsed*** and ***dwTimeEstimated*** members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

HttpEnableEvents, HttpGetTransferStatus, HttpGetTransferStatusEx, HttpRegisterEvent, HTTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Server Library

Implements a server that enables the application to access documents using the Hypertext Transfer Protocol.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSHTSV12.DLL
Version	12.0.3024.1286
LibID	25AA25CC-886B-4CFD-8089-96B70CAC2169
Import Library	CSHTSV12.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 3875

Overview

This library provides an interface for implementing an embedded, lightweight server that can be used to provide access to documents and other resources using the Hypertext Transfer Protocol. The server can accept connections from any standard web browser, third-party applications or programs developed using the SocketTools HTTP client API.

The application specifies an initial server configuration and then responds to events that are generated when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard HTTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server includes support for CGI scripting, virtual hosting, client authentication and the creation of virtual directories and files. The server also supports secure connections using TLS 1.2 and later versions. Secure connections require that a valid security certificate be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum

required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Server Functions

Function	Description
HttpAddVirtualHost	Add a new virtual host to the server virtual host table
HttpAddVirtualHostAlias	Add an alternate host name for an existing virtual host
HttpAddVirtualPath	Add a new virtual path for the specified host
HttpAddVirtualUser	Add a new virtual user for the specified host
HttpAuthenticateClient	Authenticate the client and assign access rights for the session
HttpCheckVirtualPath	Determine if the client has permission to access the specified virtual path
HttpCreateServerCredentials	Create a new server security credentials structure
HttpDeleteAllClientHeaders	Delete all of the request or response headers for the specified client session
HttpDeleteClientHeader	Delete a request or response header for the specified client session
HttpDeleteServerCredentials	Delete a previously created security credentials structure
HttpDeleteVirtualHost	Delete a virtual host associated with the specified server
HttpDeleteVirtualPath	Delete a virtual path from the specified virtual host
HttpDeleteVirtualUser	Delete a virtual user from the specified virtual host
HttpDisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
HttpEnableClientAccess	Enable or disable access rights for the specified client session
HttpEnableCommand	Enable or disable a specific server command
HttpEnumServerClients	Returns a list of active client sessions established with the specified server
HttpGetActiveClient	Return the client ID for the active client session associated with the current thread
HttpGetAllClientHeaders	Return all client request or response headers in a string buffer
HttpGetClientAccess	Return the access rights that have been granted to the client session
HttpGetClientAddress	Return the IP address of the specified client session
HttpGetClientCredentials	Return the credentials for the specified client session
HttpGetClientDirectory	Return the root document directory for a client session
HttpGetClientHeader	Return the value of a request or response header for the specified client session
HttpGetClientIdleTime	Return the idle timeout period for the specified client
HttpGetClientLocalPath	Return the full local path for the specified virtual path
HttpGetClientServer	Return the handle to the server that created the specified client session
HttpGetClientThreadId	Returns the thread ID associated with the specified client session
HttpGetClientUserName	Return the user name associated with the specified client session
HttpGetClientVariable	Return the value of a CGI environment variable for the specified client
HttpGetClientVirtualHost	Return the name of the virtual host the client used to establish the connection
HttpGetClientVirtualHostId	Return the virtual host ID associated with the specified client session
HttpGetClientVirtualPath	Return the virtual path for a local file on the server
HttpGetCommandFile	Return the full path to the local file name or directory specified by the client
HttpGetCommandLine	Return the complete command line issued by the client
HttpGetCommandName	Return the name of the command that was issued by the client

HttpGetCommandQuery	Return the query parameters included with the command
HttpGetCommandResource	Return the path for the resource requested by the client
HttpGetCommandResult	Return the result code and a description of the last command processed by the server
HttpGetCommandUrl	Return the complete URL of the resource requested by the client
HttpGetProgramExitCode	Return the exit code of the last program executed by the client
HttpGetProgramName	Return the name of the CGI program executed by the client
HttpGetProgramOutput	Return a copy of the standard output from a CGI program executed by the client
HttpGetProgramText	Return a copy of the standard output from a CGI program in a string buffer
HttpGetServerAddress	Return the IP address for the server
HttpGetServerDirectory	Return the full path to the root directory assigned to the specified server
HttpGetServerError	Return information about the last server error that occurred
HttpGetServerIdentity	Return the identity and version information for the specified server
HttpGetServerLogFile	Return the current log file format and full path for the file
HttpGetServerMemoryUsage	Return the amount of memory allocated for the server and all client sessions
HttpGetServerName	Return the host name assigned to the server or specified client session
HttpGetServerOptions	Return the configuration options for the specified server
HttpGetServerPriority	Return the current priority assigned to the specified server
HttpGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
HttpGetServerTransferInfo	Return information about the current file transfer
HttpGetServerUuid	Return the UUID assigned to the specified server
HttpGetServerUuidString	Return the UUID assigned to the server as a printable string
HttpGetVirtualHostId	Return the virtual host ID associated with the specified hostname
HttpGetVirtualHostName	Return the hostname associated with the specified virtual host ID
HttpIsClientAuthenticated	Determine if the specified client session has been authenticated
HttpIsCommandEnabled	Determine if the specified command is currently enabled or disabled
HttpReceiveRequest	Receive the request that was sent by the client to the server
HttpRedirectRequest	Redirect the request from the client to another URL
HttpRegisterHandler	Register a CGI program for use and associate it with a file name extension
HttpRegisterProgram	Register a CGI program for use and associate it with a virtual path on the server
HttpRenameServerLogFile	Rename or delete the current log file being updated by the server
HttpRequireAuthentication	Send a response to the client indicating that authentication is required
HttpSendErrorResponse	Send a customized error response to the specified client
HttpSendResponse	Send a response from the server to the specified client
HttpSendResponseData	Send additional data to the client in response to a command
HttpServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
HttpServerDisableTrace	Disable logging of network function calls
HttpServerEnableTrace	Enable logging of network function calls to a file
HttpServerInitialize	Initialize the library and validates the specified license key at runtime
HttpServerProc	Callback function used to process server events
HttpServerRestart	Restart the server, terminating all active client sessions

HttpServerResume	Resume accepting client connections on the specified server
HttpServerStart	Start the server and begin accepting client connections
HttpServerStop	Stop the server and terminate all active client connections
HttpServerSuspend	Suspend accepting client connections on the specified server
HttpServerThrottle	Limit the number of active client connections, connections per address and connection rate
HttpServerUninitialize	Terminate use of the library by the application
HttpSetClientAccess	Change the access rights associated with the specified client session
HttpSetClientHeader	Create or change the value of a request or response header for the client session
HttpSetClientIdleTime	Change the idle timeout period for the specified client session
HttpSetClientVariable	Create or change the value of a CGI environment variable for the specified client
HttpSetCommandFile	Change the name of the local file or directory that is the target of the current command
HttpSetServerError	Set the last error code for the specified server session
HttpSetServerIdentity	Change the identity and version information for the specified server
HttpSetServerLogFile	Change the current log format, level of detail and file name
HttpSetServerName	Change the hostname assigned to the specified server or client session
HttpSetServerPriority	Change the priority assigned to the specified server
HttpSetServerStackSize	Change the initial size of the stack allocated for threads created by the server.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddVirtualHost Function

```
UINT WINAPI HttpAddVirtualHost(
   HSERVER hServer,
   LPCTSTR lpszHostName,
   UINT nHostPort,
   LPCTSTR lpszDirectory
);
```

Add a new virtual host to the server virtual host table.

Parameters

hServer

The server handle.

lpszHostName

A null-terminated string which specifies the hostname that will be added to the virtual host table. This parameter must specify a valid hostname and cannot be a NULL pointer or a zero-length string.

nHostPort

An integer value which specifies the port number for the virtual host. This value must be zero or the same value as the original port number that the server was configured to use.

lpszDirectory

A null-terminated string which specifies the root document directory for the virtual host. If this parameter is NULL or a zero-length string, the virtual host will use the same root directory that was specified when the server was started. This parameter may contain environment variables enclosed in % symbols.

Return Value

If the function is successful, it will return a non-zero integer value that identifies the virtual host. If the function fails, it will return INVALID_VIRTUAL_HOST and the last error code will be updated to indicate the cause of the failure.

Remarks

Virtual hosting is a method for sharing multiple domain names on a single instance of a server. The client provides the server with the hostname that it has used to establish the connection, and that name is compared against a table of virtual hosts configured for the server. If the hostname matches a virtual host, the client will use the root directory and any virtual paths that have been assigned to that host.

When the server is first started, a default virtual host with an ID of zero is automatically created and is identified as VIRTUAL_HOST_DEFAULT. This virtual host uses the same hostname, port number and root directory that the server instance was created with. The application should treat all other host IDs as opaque values and never make assumptions about how they are allocated.

The *nHostPort* parameter should always be specified with a value of zero, or the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHostAlias, HttpAddVirtualPath, HttpAddVirtualUser, HttpDeleteVirtualHost

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddVirtualHostAlias Function

```
BOOL WINAPI HttpAddVirtualHostAlias(
    HSERVER hServer,
    UINT nHostId,
    LPCTSTR lpszHostAlias
);
```

Add an alternate host name for an existing virtual host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszHostAlias

A null-terminated string which specifies the alias for the virtual host. The alias must be a valid domain name that uniquely identifies the host. This parameter cannot be a NULL pointer or specify an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpAddVirtualHostAlias** function adds an alias for an existing virtual host. This enables a client to establish a connection using a number of different domain names which all reference the same virtual host. When the server responds to the client, it will identify itself with the primary domain name assigned to the virtual host rather than the alias provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpAddVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddVirtualPath Function

```
BOOL WINAPI HttpAddVirtualPath(
   HSERVER hServer,
   UINT nHostId,
   DWORD dwFileAccess,
   LPCTSTR lpszVirtualPath,
   LPCTSTR lpszLocalPath
);
```

Add a new virtual path for the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

dwFileAccess

An integer value which specifies the access clients will be given to the virtual path. For a list of file access permissions, see User and File Access Constants.

lpszVirtualPath

A null-terminated string which specifies the virtual path that will be created. This parameter cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

lpszLocalPath

A null-terminated string which specifies the local directory or file name that the virtual path will be mapped to. This path must exist and can be no longer than MAX_PATH characters. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpAddVirtualPath** function maps a virtual path name to a directory or file name on the local system. Virtual paths are assigned to specific hosts and if multiple virtual hosts are created for the server, each can have its own virtual paths which map to different files. To create a virtual path for the default server, the ***nHostId*** parameter should be specified as VIRTUAL_HOST_DEFAULT.

It is recommended that the ***lpszLocalPath*** parameter always specify the full path to the local file or directory. If the path is relative, it will be considered relative to the current working directory for the process and expanded to its full path name. The local path can include environment variables surrounded by % symbols. For example, if the value %ProgramData% is included in the path, it will be expanded to the full path for the common application data folder. The local path cannot specify a Windows system folder or the root directory of a mounted drive volume.

The local file or directory does not need to located in the document root directory for the server or virtual host. It can specify any valid local path that the server process has the appropriate permissions to access. You should exercise caution when creating virtual paths to files or directories outside of the

server root directory. If the ***lpszLocalPath*** parameter specifies a directory, clients will have access to that directory and all subdirectories using its virtual path.

If you wish to password protect the virtual file or directory, include the HTTP_ACCESS_PROTECTED flag in the file permissions. The default command handlers will recognize this flag and require that the client authenticate itself to grant access to the resource. If the server application implements a custom command handler, it is responsible for checking for the presence of this flag and perform the appropriate checks to ensure that the client session has been authenticated.

If the server was started in restricted mode, the client will be unable to access documents outside of the server root directory and its subdirectories. This restriction also applies to virtual paths that reference documents or other resources outside of the root directory. To allow a client to access a document outside of the server root directory, the **HttpSetClientAccess** function should be used to grant the client HTTP_ACCESS_READ permission.

The **HttpGetClientVirtualPath** function will return the virtual path that is associated with a local file or directory. The **HttpGetClientLocalPath** function will return the full path to a local file or directory that is mapped to a virtual path.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpCheckVirtualPath, HttpGetClientLocalPath, HttpGetClientVirtualPath, HttpDeleteVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAddVirtualUser Function

```
BOOL WINAPI HttpAddVirtualUser(
   HSERVER hServer,
   UINT nHostId,
   DWORD dwUserAccess,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszDirectory
);
```

Add a new virtual user for the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see User and File Access Constants.

lpszUserName

A null-terminated string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the function will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A null-terminated string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the function will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

lpszDirectory

A null-terminated string which specifies the local directory that is considered to be the virtual user's home directory. This path must exist and can be no longer than MAX_PATH characters. The maximum length of the local path is 260 characters. The directory cannot be located in a Windows system folder or the root directory of a mounted disk volume. If this parameter is NULL or an empty string, the server root directory will be assigned as the user home directory.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpAddVirtualUser** function adds a virtual user that is associated with the specified virtual host.

If a client attempts to access a protected document and provides credentials, the server will attempt to automatically authenticate the session by searching for virtual user with the same username and password. If a match is found, then the client session is assigned the same access permissions as the virtual user.

If the server is started with the HTTP_SERVER_MULTIUSER option, then documents in the virtual user's home directory can be accessed by specifying their username using a specially formatted request URL. For example, if a virtual user named "Thomas" is created, the documents in that user's home directory could be accessed as http://***servername***/~thomas/***document***.html

All files and subdirectories in the user's home directory are considered to be read-only. A client cannot create files in a user's home directory, even if they are authenticated as that user. In addition, CGI programs and scripts cannot be executed from a user's home directory.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this function is called with a username that already exists, that record is replaced with the values passed to this function.

The virtual users created by this function exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this function for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpDeleteVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpAuthenticateClient Function

```
BOOL WINAPI HttpAuthenticateClient(
    HSERVER hServer,
    UINT nClientId,
    DWORD dwUserAccess
);
```

Authenticate the client and assign access rights for the session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User and File Access Constants.

Return Value

If the client session could be authenticated, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has already been authenticated, this function will return zero.

Remarks

The **HttpAuthenticateClient** function is used to authenticate a specific client session, typically in response to an HTTP_CLIENT_USERAUTH event that indicates a client has provided authentication credentials as part of the request for a document or other resource.

To enable the server to automatically authenticate a client session, use the **HttpAddVirtualUser** function to add one or more virtual users. The server will search the list of virtual users for a match to the credentials provided by the client and will set the appropriate permissions for the session without requiring a event handler to manually authenticate the session using this function.

If the server was started with the HTTP_SERVER_LOCALUSER option and the client session is not authenticated using this function, the server will attempt to authenticate the client session using the local Windows user database. Although this option can be convenient because it does not require the implementation of an event handler for the HTTP_CLIENT_USERAUTH event, it can be used by clients to attempt to discover valid usernames and passwords for the local system. It is recommended that you use the **HttpAddVirtualUser** function to create virtual users rather than using the local user database.

It is recommended that most applications specify HTTP_ACCESS_DEFAULT as the ***dwUserAccess*** value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpAddVirtualUser, HttpGetClientCredentials, HttpGetClientDirectory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCreateServerCredentials Function

```
BOOL WINAPI HttpCreateServerCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **HttpCreateServerCredentials** function creates a SECURITYCREDENTIALS structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **HttpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The structure that is created by this function may be used as server credentials when accepting a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **HttpServerStart** function.

An application only needs to call this function to support secure connections using TLS. When a certificate is specified, it must have a private key associated with it. Attempting to use a certificate which does not have a private key will result in an error during the startup process indicating that the server credentials are invalid.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify

the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Example

```
// Create the server credentials that identifies the certificate
// that will be used for secure connections
LPSECURITYCREDENTIALS lpSecCred = NULL;

HttpCreateServerCredentials(SECURITY_PROTOCOL_DEFAULT,
                            0,
                            NULL,
                            NULL,
                            lpszCertStore,
                            lpszCertName,
                            NULL,
                            &lpSecCred);

// Start the server
hServer = HttpServerStart(lpszLocalHost,
                          HTTP_PORT_SECURE,
                          HTTP_SERVER_SECURE,
                          &httpConfig,
                          lpEventHandler,
                          0,
                          lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDeleteServerCredentials, HttpServerStart, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpCheckVirtualPath Function

```
BOOL WINAPI HttpCheckVirtualPath(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   DWORD dwFileAccess
);
```

Determine if the client has permission to access the specified virtual path.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A null-terminated string which specifies the virtual path that should be checked. This path must be absolute and cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

dwFileAccess

An unsigned integer value which specifies the access permissions that should be checked. For a list of file access permissions, see User and File Access Constants.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the client ID does not specify a valid client, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpCheckVirtualPath** function is used to determine if the client has permission to access the virtual file or directory, based on the value of the ***dwFileAccess*** parameter. For example, if the ***dwFileAccess*** parameter has the value HTTP_ACCESS_WRITE, this function will check if the client has write permission for the file or directory. The function will return a non-zero value if the client does have the requested permission, or zero if it does not.

Applications that implement their own custom handlers for standard HTTP commands should use this function to ensure that the client has the appropriate permissions to access the requested resource. Failure to check the access permissions for the client can result in the client being able to access restricted documents and other resources. It is recommended that most applications use the default command handlers.

To obtain the path to the local file or directory that the virtual path is mapped to, use the **HttpGetClientLocalPath** function.

Example

```
TCHAR szPathName[1024];

// Get the current request URL path
INT cchPathName = HttpGetCommandResource(
```

```
                    hServer,
                    nClientId,
                    szPathName,
                    1024);

if (cchPathName == 0)
{
    HttpSendErrorResponse(hServer, nClientId, 500, NULL);
    return;
}

// Check if the client has write access to that resource
BOOL bAllowed = HttpCheckVirtualPath(
                    hServer,
                    nClientId,
                    szPathName,
                    HTTP_ACCESS_WRITE);

if (!bAllowed)
{
    HttpSendErrorResponse(hServer, nClientId, 403, NULL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualPath, HttpDeleteVirtualPath, HttpGetClientLocalPath, HttpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteAllClientHeaders Function

```
BOOL WINAPI HttpDeleteAllClientHeaders(
    HSERVER hServer,
    UINT nClientId,
    UINT nHeaderType
);
```

Delete all of the request or response headers for the specified client session.

Parameters

hServer

The server handle.

nClientId

Delete all of the request or response headers for the specified client session.

nHeaderType

Specifies the type of headers to delete. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that were provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete all of the response headers that were created by the server in response to a request made by the client. Response header values provide additional information to the client about the type of information that is being returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpDeleteAllClientHeaders** function is used to delete all of the request or response headers that were set as the result of the client issuing a request for a document or other resource. If this function is used to delete all of the response headers, the server will automatically generate a standard set of response headers when it returns the requested information to the client.

It is not necessary to call this function inside an HTTP_CLIENT_DISCONNECT event handler to delete the header values that were set during the client session. This is done automatically when the client disconnects from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpDeleteClientHeader, HttpSetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteClientHeader Function

```
BOOL WINAPI HttpDeleteClientHeader(
    HSERVER hServer,
    UINT nClientId,
    UINT nHeaderType,
    LPCTSTR lpszHeaderName
);
```

Delete a request or response header for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Delete a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A null-terminated string that specifies the name of the header that should be deleted. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpDeleteClientHeader** function will delete a request or response header for the specified client session. There are a number of required response headers that are always sent to a client and deleting the header using this function will cause the server to automatically generate a new default header value. You should not delete response header values unless you are certain of the impact that it would have on the normal operation of the client.

It is not necessary for you to delete a header value to change the value of an existing header. The **HttpSetClientHeader** function will replace an existing header value with a new value, or create a new header if the header name does not already exist.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDeleteAllClientHeaders, HttpGetClientHeader, HttpSetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteServerCredentials Function

```
VOID WINAPI HttpDeleteServerCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **HttpDeleteServerCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)
    HttpDeleteServerCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated for the credentials after the server has been started.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpCreateServerCredentials, HttpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteVirtualHost Function

```
BOOL WINAPI HttpDeleteVirtualHost(
    HSERVER hServer,
    UINT nHostId
);
```

Delete a virtual host associated with the specified server.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpDeleteVirtualHost** function removes a virtual host that was created by a previous call to the **HttpAddVirtualHost** function. All virtual paths and users associated with the specified host are no longer valid. It is not necessary to call this function to delete any of the virtual hosts prior to stopping the server. Part of the normal shutdown process is releasing the resources allocated for each virtual host that was added to the server.

This function cannot be used to delete the virtual host with an ID of zero, which is the default virtual host that is allocated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpAddVirtualHost, HttpAddVirtualPath, HttpAddVirtualUser HttpDeleteVirtualPath, HttpDeleteVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteVirtualPath Function

```
BOOL WINAPI HttpDeleteVirtualPath(
   HSERVER hServer,
   UINT nHostId,
   LPCTSTR lpszVirtualPath
);
```

Remove a virtual path from the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszVirtualPath

A null-terminated string which specifies the virtual path that will be removed. This path must be absolute and cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This function removes a virtual path that was created by a previous call to the **HttpAddVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpAddVirtualPath, HttpGetClientLocalPath, HttpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDeleteVirtualUser Function

```
BOOL WINAPI HttpDeleteVirtualUser(
    HSERVER hServer,
    UINT nHostId,
    LPCTSTR lpszUserName
);
```

Remove a virtual user from the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszUserName

A null-terminated string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This function removes a virtual user that was created by a previous call to the **HttpAddVirtualUser** function. This function will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpAddVirtualUser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpDisconnectClient Function

```
BOOL WINAPI HttpDisconnectClient(
    HSERVER hServer,
    UINT nClientId
);
```

Close the control connection for the specified client and release the resources allocated for the session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpDisconnectClient** function will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted.

This function sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active data transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this function has returned.

After this function returns, the application should never use the same client ID with another function. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerRestart, HttpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnableClientAccess Function

```
BOOL WINAPI HttpEnableClientAccess(
    HSERVER hServer,
    UINT nClientId,
    DWORD dwUserAccess,
    BOOL bEnable
);
```

Enable or disable access rights for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see User and File Access Constants.

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpEnableClientAccess** function is used to enable or disable access to specific functionality by the client. The function can only change a single access right and cannot be used to enable or disable multiple access rights in a single function call. To change multiple user access rights for the client, use the **HttpSetClientAccess** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpAuthenticateClient, HttpGetClientAccess, HttpSetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnableCommand Function

```
BOOL WINAPI HttpEnableCommand(
   HSERVER hServer,
   LPCTSTR lpszCommand,
   BOOL bEnable
);
```

Enable or disable a specific server command

Parameters

hServer

The server handle.

lpszCommand

A null-terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this function. This parameter cannot be NULL.

bEnable

An integer value which specifies if the command should be enabled or disabled. If the value is non-zero, the command is enabled. If the value is zero, the command will be disabled.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the command is not recognized, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpEnableCommand** function is used to enable or disable access to a specific command on the server. This function is typically used to enable or disable certain commands for security purposes. For example, the PUT command can be disabled, preventing any client from attempting to upload files directly to the server. The **HttpIsCommandEnabled** function can be used to determine if a command is enabled or not.

The command name provided to this function must match the commands defined in RFC 2616 or related protocol standards. Refer to Hypertext Transfer Protocol Commands for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the GET and HEAD commands cannot be disabled. If you attempt to disable a required command, this function will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this function affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, use an event handler to filter the commands.

The OPTIONS and TRACE commands are disabled by default for all server instances and must be explicitly enabled using the **EnableCommand** method if you wish permit clients to use them. It is not recommended that you enable these commands if your server is going to be publicly accessible over the Internet. If the server started with the option HTTP_SERVER_READONLY, commands that can be used to create or modify files on the server will be disabled by default.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticateClient, HttpIsCommandEnabled

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpEnumServerClients Function

```
INT WINAPI HttpEnumServerClients(
    HSERVER hServer,
    DWORD dwReserved
    UINT *lpClients,
    INT nMaxClients
);
```

Return a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

dwReserved

An unsigned integer value that is reserved for future use. This parameter should always be zero.

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the function returns. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the ***lpClients*** array. If the ***lpClients*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetServerError**.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections, the function will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the function with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero.

Example

```
INT nClients = HttpEnumServerClients(hServer, 0, NULL, 0);

if (nClients > 0)
{
    UINT *lpClients = NULL;

    // Allocate memory for an array of client IDs
    lpClients = (UINT *)LocalAlloc(LPTR, nClients * sizeof(UINT));

    if (lpClients == NULL)
    {
        // Virtual memory has been exhausted
        return;
    }
```

```
    nClients = HttpEnumServerClients(hServer, 0, lpClients, nClients);

    if (nClients == HTTP_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        TCHAR szUserName[HTTP_MAXUSERNAME];

        if (HttpGetClientUserName(hServer, lpClients[nIndex], szUserName,
HTTP_MAXUSERNAME))
        {
            // Perform some action with the client user name
        }
    }

    // Free the memory allocated for the client IDs
    LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetActiveClient Function

```
UINT WINAPI HttpGetActiveClient(
    HSERVER hServer
);
```

Return the client ID for the active client session associated with the current thread.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the unique ID associated with the client session for the current thread. If the server handle is invalid or there is no client session active on the current thread, the return value is zero.

Remarks

The **HttpGetActiveClient** function is used to obtain the client ID associated with the current thread. This means this function will only return a client ID if it is called within an event handler or a function called by an event handler. If this function is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpEnumServerClients

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientAccess Function

```
BOOL WINAPI HttpGetClientAccess(
   HSERVER hServer,
   UINT nClientId,
   LPDWORD lpdwUserAccess
);
```

Return the access rights that have been granted to the client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwUserAccess

A pointer to an unsigned integer which specifies one or more access rights for the client session. For a list of user access permissions that can be granted to the client, see User and File Access Constants. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **HttpGetClientAccess** function is used to obtain all of the access rights that are currently granted to the client session. The **HttpEnableClientAccess** function can be used to enable or disable specific permissions, and the **HttpSetClientAccess** function can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;

// Check if the client has execute permission
if (HttpGetClientAccess(hServer, nClientId, &dwUserAccess))
{
    if (dwUserAccess & HTTP_ACCESS_EXECUTE)
    {
        _tprintf(_T("Client %u can execute programs and scripts\n"), nClientId);
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpAuthenticateClient, HttpEnableClientAccess, HttpSetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientAddress Function

```
INT WINAPI HttpGetClientAddress(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

Return the IP address of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A null-terminated string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientCredentials Function

```
BOOL WINAPI HttpGetClientCredentials(
   HSERVER hServer,
   UINT nClientId,
   LPHTTPCLIENTCREDENTIALS lpCredentials
);
```

Return the user credentials for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **HTTPCLIENTCREDENTIALS** structure that will contain information about the user when the function returns. This parameter cannot be NULL.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has not requested authentication, this function will return zero.

Remarks

The **HttpGetClientCredentials** function is used to obtain the username and password that was provided by the client when it requested authentication. Typically this function is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **HttpAuthenticateClient** function to specify that the session has been authenticated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticateClient, HttpRequireAuthentication, HTTPCLIENTCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientDirectory Function

```
INT WINAPI HttpGetClientDirectory(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszDirectory,
   INT nMaxLength
);
```

Returns the root document directory for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A null-terminated string buffer that will contain the root directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

This function returns the full path to the root document directory for the specified client session. If no virtual hosts have been configured, then this path will be the same as the root directory assigned to the server when it was started. If the server has been configured with multiple virtual hosts, this function will return the path to the root directory associated with the hostname provided by the client.

To convert a full path to the virtual path for a specific client session, use the **HttpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientHeader Function

```
INT WINAPI HttpGetClientHeader(
   HSERVER hServer,
   UINT nClientId,
   UINT nHeaderType,
   LPCTSTR lpszHeaderName,
   LPTSTR lpszHeaderValue,
   INT nMaxLength,
);
```

Return the value of a request or response header for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Return the value of a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Return the value of a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A null-terminated string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A buffer that will contain the header value, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the ***nMaxLength*** parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the header value buffer, including the terminating null character. If the ***lpszHeaderValue*** parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the length of the header value, not including the terminating null character. If the server handle and client ID do not specify a valid client session, or there is no header that matches the given name, the function will return zero. If the ***lpszHeaderValue***

parameter is not NULL and the buffer is not large enough to store the complete header value, the function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpGetClientHeader** function will return the value of a request or response header for the specified client session. If the *lpszHeaderName* value matches an existing header field, its value will be copied to the string buffer provided by the caller.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDeleteClientHeader, HttpGetAllClientHeaders, HttpSetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientIdleTime Function

```
UINT WINAPI HttpGetClientIdleTime(
    HSERVER hServer,
    UINT nClientId,
    UINT *lpnElapsed
);
```

Return the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

A pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL if this information is not required.

Return Value

If the function succeeds, the return value is client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetClientIdleTime** function will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed. The server will never disconnect a client that is in the process of sending or receiving data, regardless of the idle timeout period.

The default idle timeout period for a client session is 60 seconds, however the server can be configured to use a different value. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). An application can change the timeout period for a specific client session using the **HttpSetClientIdleTime** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpSetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientLocalPath Function

```
INT WINAPI HttpGetClientLocalPath(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszVirtualPath,
   LPTSTR lpszLocalPath,
   INT nMaxLength,
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A null-terminated string that specifies an virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A null-terminated string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **HttpGetClientLocalPath** function takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the root directory for the client session.

To obtain the virtual path for a local file or directory, use the **HttpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientVirtualPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientServer Function

```
HSERVER WINAPI HttpGetClientServer(
     UINT nClientId
);
```

The **HttpGetClientServer** function returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is INVALID_SERVER. To get extended error information, call the **HttpGetServerError** function.

Remarks

The **HttpGetClientServer** function returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this function will fail and return INVALID_SERVER indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerAsyncNotify

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientThreadId Function

```
DWORD WINAPI HttpGetClientThreadId(
    HSERVER hServer,
    UINT nClientId
);
```

Returns the thread ID associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is a thread ID. If the function fails, the return value is zero. To get extended error information, call the **HttpGetServerError** function.

Remarks

The **HttpGetClientThreadId** function returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this function to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **HttpDisconnectClient** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpEnumServerClients, HttpGetActiveClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientUserName Function

```
INT WINAPI HttpGetClientUserName(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszUserName,
   INT nMaxLength
);
```

Return the user name associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A null-terminated string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value larger than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetClientUserName** function returns the user name associated with an authenticated client session. If the client has not authenticated itself, this function will return zero and the *lpszUserName* parameter will be set to an empty string.

The **HttpIsClientAuthenticated** function can be used to determine if the client has provided credentials as part of the request made to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticateClient, HttpGetClientAccess, HttpIsClientAuthenticated

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientVariable Function

```
INT WINAPI HttpGetClientVariable(
    HSERVER hServer,
    UINT nClientId,
    LPCTSTR lpszName,
    LPTSTR lpszValue,
    INT nMaxLength,
);
```

Return the value of a CGI environment variable for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A null-terminated string that specifies the name of the environment variable. Variable names are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A buffer that will contain the value of the environment variable, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the ***nMaxLength*** parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the value buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the length of the environment variable value, not including the terminating null character. If the server handle and client ID do not specify a valid client session, or there is no environment variable that matches the given name, the function will return zero. If the ***lpszValue*** parameter is not NULL and the buffer is not large enough to store the complete header value, the function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpGetClientVariable** function will return the value of an environment variable that has been defined for the client. Each client session inherits a copy of the process environment block, which is then modified to define various environment variables that are used with CGI programs and scripts. The **HttpSetClientVariable** function can be used to change existing environment variables or create new variables.

The standard CGI environment variables that are defined by the server are not created until the client request has been processed. This means that environment variables such as REMOTE_ADDR and SERVER_NAME will not be defined inside an HTTP_CLIENT_CONNECT event handler.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpSetClientVariable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientVirtualHost Function

```
INT WINAPI HttpGetClientVirtualHost(
    HSERVER hServer,
    UINT nClientId,
    LPTSTR lpszHostName,
    INT nMaxLength
);
```

Return the name of the virtual host the client used to establish the connection.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A null-terminated string buffer that will contain the virtual host name. The string buffer will be null-terminated and must be large enough to store the complete hostname. If this parameter is NULL, the function will only return the length of the current command in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the ***lpszHostName*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetClientVirtualHost** function is used to obtain the hostname that the client used to establish a connection with the server. This function is typically used within an event handler to determine the hostname associated with the request made by the client. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the virtual host has not been selected at that point. If the virtual hostname is not available at the time this function is called, the function will return zero and the last error code will be set to ST_ERROR_VIRTUAL_HOST_NOT_FOUND.

The **HttpGetClientVirtualHostId** function can be used to obtain the virtual host ID associated with the hostname.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientVirtualHostId, HttpGetCommandUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientVirtualHostId Function

```
UINT WINAPI HttpGetClientVirtualHostId(
    HSERVER hServer,
    UINT nClientId
);
```

Return the virtual host ID associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An unsigned integer value which specifies the virtual host ID. If the function fails, the return value will be INVALID_VIRTUAL_HOST and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetClientVirtualHostId** function is used to obtain the virtual host ID associated with the hostname the client used to establish a connection with the server. This function should not be called inside a HTTP_CLIENT_CONNECT event handler because the virtual host has not been selected at that point. If the virtual host ID is not available at the time this function is called, the function will return INVALID_VIRTUAL_HOST and the last error code will be set to ST_ERROR_VIRTUAL_HOST_NOT_FOUND.

The **HttpGetClientVirtualHost** function can be used to obtain the hostname that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetClientVirtualHost, HttpGetCommandUrl, HttpGetVirtualHostId, HttpGetVirtualHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetClientVirtualPath Function

```
INT WINAPI HttpGetClientVirtualPath(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszLocalPath,
   LPTSTR lpszVirtualPath,
   INT nMaxLength,
);
```

Return the virtual path for a local file on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A null-terminated string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A null-terminated string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is relative to the root directory for the specified client session. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the ***lpszLocalPath*** parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this function would return the virtual path to that directory as "/Documents/Research".

If the ***lpszLocalPath*** parameter specifies a file or directory outside of the server root directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This function can only be used with authenticated clients. If the ***nClientId*** parameter specifies a client session that has not been authenticated, this function will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandFile, HttpGetClientLocalPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandFile Function

```
INT WINAPI HttpGetCommandFile(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

Get the full path to a file name or directory specified by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null-terminated and must be large enough to store the complete file path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetCommandFile** function is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard HTTP command. For example, if the client sends the GET command to the server, this function will return the complete path to the local file that the client wants to retrieve. This function will only work with those standard commands that perform some action on a file or directory.

This function should always be used to obtain the file name for a command that performs a file or directory operation. It normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location. The **HttpGetCommandUrl** function can be used to obtain the URL that was provided by the client.

To map a virtual path to a file or directory on the local system, use the **HttpAddVirtualPath** function. To redirect a client to use a different URL to access the resource, use the **HttpRedirectRequest** function.

The **HttpSetCommandFile** function can be used to change the name of the local file or directory that is the target of the command, however using this function to redirect access to a resource can have unintended side-effects, particularly in the case where the URL provided by the client actually resolves to an executable CGI program that handles the request.

If the client has provided a URL that resolves to a CGI program that handles the request, this function behaves differently than if the URL is resolved to a local file or directory. If the client uses the GET or POST command that results in a program being executed to handle the request, this function will return the path to the server root directory along with any additional path information provided in the URL. In other words, the file name returned by this function will be the same as the PATH_TRANSLATED value passed to the CGI program.

This function should only be called after the client request has been received by the server, typically inside a HTTP_CLIENT_COMMAND or HTTP_CLIENT_EXECUTE event handler. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualPath, HttpGetCommandUrl, HttpRedirectRequest, HttpSetCommandFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandLine Function

```
INT WINAPI HttpGetCommandLine(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszCmdLine,
   INT nMaxLength
);
```

Return the complete command line issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A null-terminated string buffer that will contain the command, including all arguments. The string buffer will be null-terminated and must be large enough to store the complete command line. If this parameter is NULL, the function will return the length of the command line.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **HttpGetCommandLine** function is used to obtain the command that was issued by the client, and is commonly used inside HTTP_CLIENT_COMMAND and HTTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively. When the function returns, the string buffer provided by the caller will contain the complete command, including the resource path and requested HTTP version. Any extraneous whitespace will be removed, however all encoding will be preserved.

To obtain the complete URL associated with the request issued by the client, use the **HttpGetCommandUrl** function. If the command sent by the client is used to perform an action on a file or directory, the **HttpGetCommandFile** function should be called to obtain the full path to the local file rather than using the resource path.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandFile, HttpGetCommandUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandName Function

```
INT WINAPI HttpGetCommandName
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszCommand,
   INT nMaxLength
);
```

Return the name of the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A null-terminated string buffer that will contain the command name. The string buffer will be null-terminated and must be large enough to store the complete command name. If this parameter is NULL, the function will only return the length of the current command in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the ***lpszCommand*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **HttpGetCommandName** function is used to obtain the name of the last command that was issued by the client. The command name returned by this function will always be capitalized, regardless of how it was sent by the client. This function is typically used inside HTTP_CLIENT_COMMAND and HTTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

The **HttpGetCommandUrl** function can be used to return the resource that was requested by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandFile, HttpGetCommandUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandQuery Function

```
INT WINAPI HttpGetCommandQuery(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszParameters,
   INT nMaxLength
);
```

Return the query parameters included with the command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszParameters

A null-terminated string buffer that will contain the query parameters when the function returns. The string buffer will be null terminated up to the maximum number of characters specified by the caller. If this parameter is NULL the function will return the length of the query string.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the ***lpszParameters*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code. If the client did not provide any query parameters, this function will return zero and the last error code will be zero.

Remarks

The **HttpGetCommandQuery** function is used to obtain a copy of the query parameters that were included in the request URL. If there were no query parameters, the string buffer will be empty and the return value will be zero. If the request did include query parameters, they will be returned to the caller in their original, encoded form.

This function should not be called within a HTTP_CLIENT_CONNECT event handler because the client request has not been processed at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandName, HttpGetCommandResource, HttpReceiveRequest

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandResource Function

```
INT WINAPI HttpGetCommandResource(
    HSERVER hServer,
    UINT nClientId,
    LPTSTR lpszResource,
    INT nMaxLength
);
```

Return the URL path for the resource requested by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResource

A null-terminated string buffer that will contain the URL path provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete path. If this parameter is NULL, the function will only return the length of the path.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the ***lpszResource*** parameter is NULL, this value should be zero. If this value is less than the length of the URL, the function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetCommandResource** function returns the URL path that the client provided to access the requested resource. The path will not include the URI scheme, user credentials or query parameters. The **HttpGetCommandFile** function can be used to determine the name of the local file on the server that will be accessed using this URL.

If you require the complete URL, not just the path to the resource, use the **HttpGetCommandUrl** function.

This function should only be called after the client request has been received by the server, typically inside a HTTP_CLIENT_COMMAND event handler. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandFile, HttpGetCommandQuery, HttpGetCommandUrl, HttpRedirectRequest

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandResult Function

```
INT WINAPI HttpGetCommandResult(
    HSERVER hServer,
    UINT nClientId,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

Return the result code and description for the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A null-terminated string buffer that will contain the description of the result code. The string buffer will be null-terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the ***lpszResult*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **HttpGetCommandResult** function is used to determine the result of the last command that was issued by the client and is typically called in the HTTP_CLIENT_RESULT event handler. This function should only be called after a command has been processed or the **HttpSendResponse** function has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code and a brief description of the result. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range are informational and only used with version 1.1 of the protocol. If the server returns a result code in this range, it means that it has received the request and that the client should proceed.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. In most cases this means that the server has returned the requested data to the client, however if a 204 result code is sent, this indicates that the request has been processed but there is no data available.
300-399	Result codes in this range indicate that the requested resource has been moved to a

	new location. The most common result codes are 301 and 302. A value of 301 indicates that the location of the resource has changed permanently and all future requests should be sent to the new URL. A value of 302 indicates that the resource location has changed temporarily. The new location of the resource is sent to the client by setting the Location response header field.
400-499	Result codes in this range indicate an error with the request that was made by the client. These types of errors include invalid commands, requests that can only be issued by authenticated clients, or resources that cannot be accessed on the server. The most common result code in this range is the 404 code which indicates that the requested document could not be found.
500-599	Result codes in this range indicate a server error has occurred while processing a valid request. These types of errors are returned when a command has not been implemented, the execution of a CGI program or script has failed unexpectedly, or an internal server error has occurred.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpReceiveRequest, HttpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetCommandUrl Function

```
INT WINAPI HttpGetCommandUrl(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszCmdUrl,
   INT nMaxLength
);
```

Return the complete URL of the resource requested by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdUrl

A null-terminated string buffer that will contain the URL provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete URL. If this parameter is NULL, the function will only return the length of the URL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the ***lpszCmdUrl*** parameter is NULL, this value should be zero. If this value is less than the length of the URL, the function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetCommandUrl** function returns the complete URL that the client provided to access the requested resource. The URL will include any query parameters that were specified by the client, but it will not include any user credentials. The **HttpGetCommandFile** function can be used to determine the name of the local file on the server that will be accessed using this URL.

If you only require the URL path, without the URI scheme or the query parameters, use the **HttpGetCommandResource** function.

This function should only be called after the client request has been received by the server, typically inside a HTTP_CLIENT_COMMAND event handler. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandFile, HttpGetCommandResource, HttpRedirectRequest

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetProgramExitCode Function

```
BOOL WINAPI HttpGetProgramExitCode(
    HSERVER hServer,
    UINT nClientId,
    LPDWORD lpdwExitCode
);
```

Return the exit code of the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwExitCode

A pointer to an unsigned integer that will contain the program exit code when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetProgramExitCode** function returns the exit code of a registered CGI program or script that was executed. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by programs to communicate status information back to the server application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetProgramOutput, HttpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetProgramName Function

```
INT WINAPI HttpGetProgramName(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszProgramName,
   INT nMaxLength
);
```

Return the name of the CGI program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A null-terminated string buffer that will contain the name of the CGI program executed by the client. This parameter cannot be NULL and should be at least MAX_PATH characters in size.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the function succeeds, the return value is the length of the executable file name. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed a CGI program this function will return zero and the last error code will be set to zero.

Remarks

The **HttpGetProgramName** function returns the local file name of the CGI program that was executed in response to a request from the client. This is the full path to the executable that was registered with the server using the **HttpRegisterProgram** function. If the client specified a regular document or directory, this function will return a value of zero, indicating that no CGI program has been executed to handle the request.

To obtain the resource URL that was provided by the client, use the **HttpGetCommandUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetProgramExitCode, HttpGetProgramOutput, HttpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetProgramOutput Function

```
DWORD WINAPI HttpGetProgramOutput(
    HSERVER hServer,
    UINT nClientId,
    LPBYTE lpBuffer,
    DWORD dwBufferSize
);
```

Return a copy of the standard output from a CGI program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the function will return the number of bytes of data that was output by the program. Note that this output is not null-terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the ***lpBuffer*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of bytes copied into the specified buffer. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **HttpGetProgramOutput** function is used to obtain a copy of the output generated by a CGI program. To determine the number of bytes of output available to read, call this function with the ***lpBuffer*** parameter as NULL and the ***dwBufferSize*** parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null terminated.

This function returns the raw output from the program which may contain a response header block, escape sequences, control characters and embedded nulls. When the application processes the output returned by this function, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the program as a string, use the **HttpGetProgramText** function.

This function should only be used within an HTTP_CLIENT_EXECUTE event handler, which occurs after the program has terminated.

Example

```
LPBYTE lpBuffer = NULL;  // The output buffer
DWORD cbBuffer = 0;      // Number of bytes in the output buffer

// Determine the number of bytes in the output buffer
```

```
cbBuffer = HttpGetProgramOutput(hServer, nClientId, NULL, 0);

if (cbBuffer > 0)
{
    // Allocate memory for the buffer
    lpBuffer = (LPBYTE)LocalAlloc(LPTR, cbBuffer + 1);

    // Copy the program output to the buffer
    if (lpBuffer != NULL)
        cbBuffer = HttpGetProgramOutput(hServer, nClientId, lpBuffer, cbBuffer +
1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    LocalFree((HLOCAL)lpBuffer);
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetProgramExitCode, HttpGetProgramText, HttpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetProgramText Function

```
INT WINAPI HttpGetProgramText(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

Return a copy of the standard output from a CGI program in a string buffer.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the function will return the number of bytes of characters that was output by the program, not including a terminating null character.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the ***lpszBuffer*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **HttpGetProgramText** function is used to obtain a copy of the output generated by a CGI program. To determine the number of characters of output available to read, call this function with the ***lpszBuffer*** parameter as NULL and the ***nMaxLength*** parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This function will only return textual output from the program and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the program, use the **HttpGetProgramOutput** function. If the program outputs a response header block, this will be included in the string buffer.

This function should only be used within an HTTP_CLIENT_EXECUTE event handler, which occurs after the program has terminated.

Example

```
LPTSTR lpszBuffer = NULL;  // The output buffer
INT cchBuffer = 0;         // Number of bytes in the output buffer
```

```
// Determine the number of bytes in the output buffer
cchBuffer = HttpGetProgramText(hServer, nClientId, NULL, 0);

if (cchBuffer > 0)
{
    // Allocate memory for the string buffer
    lpszBuffer = (LPTSTR)LocalAlloc(LPTR, (cchBuffer + 1) * sizeof(TCHAR));

    // Copy the program output to the buffer
    if (lpszBuffer != NULL)
        cchBuffer = HttpGetProgramText(hServer, nClientId, lpszBuffer, cchBuffer +
1);
}

// Free the memory allocated for the buffer when finished
if (lpszBuffer != NULL)
{
    LocalFree((HLOCAL)lpszBuffer);
    lpszBuffer = NULL;
    cchBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetProgramExitCode, HttpGetProgramOutput, HttpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerAddress Function

```
INT WINAPI HttpGetServerAddress(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

Return the IP address of the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A null-terminated string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

This function will return the IP address assigned to the specified server as a printable string. If the ***nClientId*** parameter has a value of zero, this function will return the IP address assigned to the local system. If the HTTP_SERVER_NATROUTER option was specified when the server was started, this function will return the external IP address assigned to the system. If the ***nClientId*** parameter specifies a valid client session, this function will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **HttpGetClientAddress** function.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientAddress, HttpGetServerName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerDirectory Function

```
INT WINAPI HttpGetServerDirectory(
   HSERVER hServer,
   LPTSTR lpszDirectory,
   INT nMaxLength
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

hServer

The server handle.

lpszDirectory

A null-terminated string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **HttpGetServerDirectory** function will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, a temporary root directory will be created and this function can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This function will return the fully resolved path name, with all environment variables expanded.

There is no corresponding function to change the server root directory after the server has started. To change the root directory, you must stop the server using the **HttpServerStop** function and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerAddress, HttpGetServerIdentity, HttpGetServerName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerError Function

```
DWORD WINAPI HttpGetServerError(
    HSERVER hServer,
    LPTSTR lpszError,
    INT nMaxLength
);
```

Return the last server error code and a description of the error.

Parameters

hServer

The server handle.

lpszError

A null-terminated string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the ***lpszError*** parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **HttpGetServerError** function immediately when a function's return value indicates that an error has occurred. That is because some functions clear the last error code when they succeed.

If the ***hServer*** parameter is specified with a value of INVALID_SERVER, this function will return the last error that occurred for the current thread. This value should only be used when the function does not have access to a valid server handle, such as when the **HttpServerStart** function fails.

It is important to note that the error codes returned by this method are different than the command result codes that are defined in RFC 2616, the standard protocol specification for HTTP. This function is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **HttpSendResponse** function is used to send responses to the client, and the **HttpGetCommandResult** function can be used to determine the result of the last command sent by the client.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR. Those functions which clear the last error code when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandResult, HttpSendResponse, HttpSetServerError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerIdentity Function

```
INT WINAPI HttpGetServerIdentity(
   HSERVER hServer,
   LPTSTR lpszIdentity,
   INT nMaxLength
);
```

Return the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A null-terminated string buffer that will contain the identity of the server when the function returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **HttpGetServerIdentity** function returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number. The **HttpSetServerIdentity** function can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpSetServerIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerLogFile Function

```
BOOL WINAPI HttpGetServerLogFile(
   HSERVER hServer,
   UINT *lpnLogFormat,
   UINT *lpnLogLevel,
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

Return the current log file format and the full path to the file.

Parameters

hServer

The server handle.

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the function returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Value	Description
HTTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A null-terminated string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the ***lpszFileName*** parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **HTTPSERVERCONFIG** structure definition for a list of supported log file formats. If the server handle is invalid or logging has not been enabled, this function will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This function can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **HttpRenameServerLogFile** function to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **HttpSetServerLogFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpRenameServerLogFile, HttpSetServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerMemoryUsage Function

```
BOOL WINAPI HttpGetServerMemoryUsage(
   HSERVER hServer,
   ULARGE_INTEGER *lpMemUsage
);
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

hServer

The server handle.

lpMemUsage

A pointer to a ULARGE_INTEGER variable which will specify how much memory has been allocated by the server. This parameter will be initialized to a value of zero by the function and updated with the total number of bytes allocated by the server and all active client sessions when it returns.

Return Value

If the function succeeds, the return value is non-zero and the memory usage value will be updated. If the server handle is invalid, or the server cannot be locked, the return value is zero. Call the **HttpGetServerError** function to determine the cause of the failure.

Remarks

This function returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This function forces the server into a locked state, and all client sessions will block until the function returns. Because this function enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this function can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the function remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetServerStackSize, HttpSetServerStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerName Function

```
INT WINAPI HttpGetServerName(
   HSERVER hServer,
   UINT nClientId,
   LPTSTR lpszHostName,
   INT nMaxLength
);
```

Return the host name assigned to the specified server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will return the host name assigned to the specified server. If the ***nClientId*** parameter has a value of zero, the function will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the ***nClientId*** parameter specifies a client session, then it this function will return the host name that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerOptions Function

```
BOOL WINAPI HttpGetServerOptions(
    HSERVER hServer,
    LPDWORD lpdwOptions
);
```

Return the configuration options for the specified server.

Parameters

hServer

The server handle.

lpdwOptions

A pointer to an unsigned integer that will contain the options that were specified when the server was started. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

This function will return the options that were specified when the server was started. For a list of server options see Server Option Constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerPriority Function

```
INT WINAPI HttpGetServerPriority(
    HSERVER hServer
);
```

Return the current priority assigned to the specified server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpGetServerPriority** function can be used to determine the current priority assigned to the server. It will return one of the following values:

Value	Description
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerStart, HttpSetServerPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerStackSize Function

```
DWORD WINAPI HttpGetServerStackSize(
     SOCKET hServer
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetServerMemoryUsage, HttpServerStart, HttpSetServerStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerTransferInfo Function

```
BOOL WINAPI HttpGetServerTransferInfo(
   HSERVER hServer,
   UINT nClientId,
   LPHTTPSERVERTRANSFER lpTransferInfo
);
```

Return information about the current file transfer.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **HTTPSERVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the ***dwSize*** member of the structure must be initialized to specify the structure size prior to calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetServerTransferInfo** function is used to obtain information about the last file transfer that was performed by the server for the specified client. This function is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

This function will only return information about a file transfer using the GET or PUT commands, with any other command causing this function to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpReceveRequest, HttpSendResponse, HTTPSERVERTRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerUuid Function

```
INT WINAPI HttpGetServerUuid(
    HSERVER hServer,
    UUID *lpUuid
);
```

Return the UUID assigned to the specified server.

Parameters

hServer

The server handle.

lpUuid

A pointer to a UUID structure that will contain the server UUID when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle is invalid or the pointer to the UUID structure is NULL, the function will return a value of zero.

Remarks

The **HttpGetServerUuid** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain a printable string version of the UUID, use the **HttpGetServerUuidString** function.

There is no corresponding function to change the UUID assigned to an active server. The server UUID is assigned when the server is started, and it must be a unique value that is maintained throughout the lifetime of the server. To change the UUID associated with the server, the server must be stopped using the **HttpServerStop** function and another instance of the server started with the new UUID.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetServerUuidString, HTTPSERVERCONFIG

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetServerUuidString Function

```
INT WINAPI HttpGetServerUuidString(
    HSERVER hServer,
    LPTSTR lpszHostUuid,
    INT nMaxLength
);
```

Return the UUID assigned to the server as a printable string.

Parameters

hServer

The server handle.

lpszHostUuid

A null-terminated string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the buffer is not large enough to store the complete UUID string, the function will return a value of zero.

Remarks

The **HttpGetServerUuidString** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain the numeric UUID value, use the **HttpGetServerUuid** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerUuid, HTTPSERVERCONFIG

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetVirtualHostId Function

```
UINT WINAPI HttpGetVirtualHostId(
   HSERVER hServer,
   LPCTSTR lpszHostName,
   UINT nHostPort
);
```

Return the virtual host ID associated with the specified hostname.

Parameters

hServer

The server handle.

lpszHostName

A string that specifies the virtual host name.

nHostPort

An integer value which specifies the port number for the virtual host. This value must be zero or the same value as the original port number that the server was configured to use.

Return Value

If the function succeeds, the return value is the host ID that uniquely identifies the virtual host. If the server handle is invalid, or there is no virtual host with the specified name, the function will return VIRTUAL_HOST_UNKNOWN. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpGetVirtualHostId** function is used to obtain the unique virtual host ID that is associated with a specific hostname. This function will match both the primary virtual hostname added using the **HttpAddVirtualHost** function, as well as any aliases that were added using the **HttpAddVirtualHostAlias** function. To obtain the virtual host ID associated with the active client session, use the **HttpGetClientVirtualHostId** function.

The *nHostPort* parameter should always be specified with a value of zero, or the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpAddVirtualHostAlias, HttpDeleteVirtualHost, HttpGetClientVirtualHostId, HttpGetVirtualHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpGetVirtualHostName Function

```
INT WINAPI HttpGetVirtualHostName(
    HSERVER hServer,
    UINT nHostId,
    LPTSTR lpszHostName,
    INT nMaxLength
);
```

Return the hostname associated with the specified virtual host ID.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszHostName

A null-terminated string buffer that will contain the virtual host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. If this parameter is NULL, the function will return the length of the virtual hostname.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. If the ***lpszHostName*** parameter is NULL this value must be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the host ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

The **HttpGetVirtualHostName** function returns the primary hostname associated with the specified virtual host ID. This is the same hostname that was specified when the virtual host was added to the server configuration using the **HttpAddVirtualHost** function. To obtain the hostname that was used by the active client session to connect to the server, use the **HttpGetClientVirtualHost** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualHost, HttpAddVirtualHostAlias, HttpDeleteVirtualHost, HttpGetClientVirtualHost, HttpGetVirtualHostId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpIsClientAuthenticated Function

```
BOOL WINAPI HttpIsClientAuthenticated(
    HSERVER hServer,
    UINT nClientId
);
```

Determine if the specified client session has been authenticated.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this function will return a non-zero value, otherwise it will return zero. If the server handle and client ID are valid, this function will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpAuthenticateClient, HttpGetClientCredentials, HttpRequireAuthentication

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpIsCommandEnabled Function

```
BOOL WINAPI HttpIsCommandEnabled(
    HSERVER hServer,
    LPCTSTR lpszCommand
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

hServer

The server handle.

lpszCommand

A null-terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name. Partial matches are not recognized by this function. This parameter cannot be NULL.

Return Value

If the command is enabled, this function will return a non-zero value. If the command is disabled, the server handle is invalid or the command name does not match a supported command, this function will return zero.

Remarks

The **HttpIsCommandEnabled** function is used to determine whether a specific command is enabled. Typically this function is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled and disabled using the **HttpEnableCommand** function.

This function does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **HttpAuthenticateClient** function, and certain commands can be limited by the permissions granted to the client. For example, even if the PUT command is enabled, a client must have the HTTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see User Access Constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAuthenticateClient, HttpEnableCommand, HttpGetCommandName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpReceiveRequest Function

```
BOOL WINAPI HttpReceiveRequest(
   HSERVER hServer,
   UINT nClientId,
   DWORD dwOptions,
   LPHTTPREQUEST lpRequest,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength
);
```

Receive the request that was sent by the client to the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be stored. It may be one of the following values:

Value	Description
HTTP_REQUEST_DEFAULT	If the ***lpdwLength*** parameter points to a integer with a non-zero value, the ***lpvBuffer*** parameter is considered to be a pointer to a block of memory that has been allocated to store the request data. This is the same as specifying the HTTP_REQUEST_MEMORY option. If the ***lpdwLength*** parameter points to an integer with a value of zero, the ***lpvBuffer*** parameter is considered to be a pointer to an HGLOBAL memory handle. This is the same as specifying the HTTP_REQUEST_HGLOBAL option.
HTTP_REQUEST_MEMORY	The ***lpvBuffer*** parameter is a pointer to a block of memory that has been allocated to store the request data. The maximum number of bytes of data that can stored is determined by the value of the integer that the ***lpdwLength*** parameter points to. When the function returns, that value will updated with with actual number of bytes copied into the buffer.
HTTP_REQUEST_STRING	The ***lpvBuffer*** parameter is a pointer to a string buffer that has been allocated to store the request data. The maximum number of bytes of data that can be copied is determined by the value of the integer that the ***lpdwLength*** parameter points to. When the function returns, that value will updated with with actual number of bytes copied into the buffer. If the Unicode version of this function is called, the request data will be converted to a null-terminated Unicode string.
HTTP_REQUEST_HGLOBAL	The ***lpvBuffer*** parameter is a pointer to an HGLOBAL memory

	handle. When the function returns, the handle will reference a block of memory that contains the request data submitted by the client. The *lpdwLength* parameter will contain the number of bytes copied to the buffer.
HTTP_REQUEST_FILE	The *lpvBuffer* parameter is a pointer to a string which specifies the name of a file that will contain the request data. If the file does not exist, it will be created. If it does exist, the contents will be replaced. This option is typically used in conjunction with the PUT command. If the *lpdwLength* parameter is not NULL, the value it points to will be updated with the actual number of bytes stored in the file.
HTTP_REQUEST_HANDLE	The *lpvBuffer* parameter is a handle to an open file. This option is typically used in conjunction with the POST or PUT commands. If the *lpdwLength* parameter is not NULL, the value it points to will be updated with the actual number of bytes written to the file. If this option is specified, the request data will be written from the current position in the file and will advance the file pointer by the number of bytes received from the client.

lpRequest

A pointer to a **HTTPREQUEST** structure which contains information about the request from the client. This parameter cannot be NULL. The structure that is passed to this function must have all members set to a value of zero except the *dwSize* member, which must be initialized to the size of the structure.

lpvBuffer

The buffer that will contain any request data that was submitted by the client. The *dwOptions* parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. If this parameter is NULL, any data submitted by the client will not be copied.

lpdwLength

A pointer to an unsigned integer that will contain the number of bytes of data submitted by the client when the function returns. If the *lpvBuffer* parameter specifies a memory or string buffer, this value must be initialized to the maximum size of the buffer before the function is called. If the *lpvBuffer* parameter points to a global memory handle, this value must be initialized to zero. If *lpvBuffer* is NULL or specifies a file name, this parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpReceiveRequest** function is called within a HTTP_CLIENT_COMMAND event handler to process the command issued by the client and return information about the request to the server application. It is only necessary for the application to call this function if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

This function may only be called once per command issued by the client and the data referenced in the **HTTPREQUEST** structure only remains valid while the client is connected to the server. You should never attempt to directly modify the data referenced by any of the structure members. If you wish to store or modify any of the string values returned in the structure, you should allocate a buffer large enough to store the contents of the string, including the terminating null character, and copy the string into that buffer.

If the HTTP_REQUEST_HGLOBAL option is used to return a copy of the request data in a global memory buffer, the HGLOBAL handle must be freed by the application when the data is no longer needed. Failure to free this handle will result in a memory leak.

If the HTTP_REQUEST_HANDLE option is used to write a copy of the request data to an open file, the handle must reference a disk file that was opened or created using the **CreateFile** function with GENERIC_WRITE access. It cannot be a handle to a device or named pipe. If the function succeeds, the file pointer is advanced by the number of bytes of request data submitted by the client. If the function fails, the file pointer is returned to its original position prior to the function being called.

Example

```
// Initialize the HTTPREQUEST structure
HTTPREQUEST httpRequest;
ZeroMemory(&httpRequest, sizeof(httpRequest));
httpRequest.dwSize = sizeof(httpRequest);

// Return the data in a global memory buffer
HGLOBAL hgblBuffer = NULL;
DWORD dwLength = 0;

bSuccess = HttpReceiveRequest(hServer,
                              nClientId,
                              HTTP_REQUEST_HGLOBAL,
                              &httpRequest,
                              &hgblBuffer,
                              &dwLength);

if (bSuccess && hgblBuffer != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    if (lpBuffer != NULL)
    {
        // Process dwLength bytes of data submitted by the client
    }

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpSendResponse, HTTPREQUEST

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRedirectRequest Function

```
BOOL WINAPI HttpRedirectRequest(
   HSERVER hServer,
   UINT nClientId,
   UINT nMethod,
   LPCTSTR lpszLocation
);
```

Redirect the request from the client to another URL.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nMethod

An integer value that specifies if the redirection is permanent or temporary. The following values may be used:

Value	Description
HTTP_REDIRECT_PERMANENT	This value is used for permanent redirection, indicating that the client should update any record of the link with the new URL specified by the ***lpszLocation*** parameter. This result is cacheable and when the client makes subsequent requests for the resource, it should always use the new URL.
HTTP_REDIRECT_TEMPORARY	This value is used for temporary redirection, indicating that the client should issue a request for the resource using the new URL specified by the ***lpszLocation*** parameter, but subsequent requests should continue to use the original URL.
HTTP_REDIRECT_OTHER	This value is used for temporary redirection, however it instructs the client that it should use the GET command to request the redirected resource. This option is typically used to redirect a client after it has used the POST command.

lpszLocation

A null-terminated string that specifies the new location for the requested resource. This value must be a complete URL, including the http:// or https:// scheme. This parameter cannot be NULL or point to zero-length string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRedirectRequest** function can be used within a HTTP_CLIENT_COMMAND event handler to redirect the client to a new location for the resource that it has requested. This redirection can be permanent or temporary, depending on whether the server expects the client to continue to use the original URL when requesting the resource.

If the HTTP_REDIRECT_TEMPORARY method is used, the actual status code that is returned to the client depends on the version of the protocol that is being used. If the client has issued the request using HTTP 1.0 then the server will return a 302 code to the client. If the client used HTTP 1.1, the server will return a 307 code to the client that indicates it should use the same command verb (GET, POST, etc.) when requesting the resource at the new location.

If the HTTP_REDIRECT_OTHER method is used, the status code that is returned to the client depends on which version of the protocol is being used. For clients who are using HTTP 1.0, the server will return a 302 code to the client just as with the HTTP_REDIRECT_TEMPORARY method. If the client is using HTTP 1.1, the server will return a 303 code to the client that indicates it should always use the GET command to request the new resource, regardless if a different command was originally used (POST, PUT, etc.)

This function provides a simplified interface for sending a redirection status code that also implicitly sets the Location response header to the value of the ***lpszLocation*** parameter. If the server application needs to send alternate redirection codes such as 305 (Use Proxy) then it should use the **HttpSendReponse** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpRequireAuthentication, HttpSendErrorResponse, HttpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRegisterHandler Function

```
BOOL WINAPI HttpRegisterHandler(
   HSERVER hServer,
   UINT nHostId,
   LPCTSTR lpszExtension,
   LPCTSTR lpszProgramFile,
   LPCTSTR lpszParameters,
   LPCTSTR lpszDirectory
);
```

Register a CGI program for use and associate it with a file name extension.

Parameters

hServer

The server handle.

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszExtension

A null-terminated string which specifies the file name extension that is associated with the CGI program. This parameter cannot be NULL.

lpszProgramFile

A null-terminated string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A null-terminated string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A null-terminated string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRegisterHandler** function registers an executable CGI program and associates it with a file name extension. When the client issues a GET or POST command that specifies a file with that extension, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple file name extensions that reference the same program. The only requirement is that the extension be unique for the given

host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the full path to the local script filename. If no placeholder is included in the parameters, or *lpszParameters* is a NULL pointer, the script file name will be passed to the program as its only argument.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **HttpGetProgramOutput** function from within a HTTP_CLIENT_EXECUTE event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
// Register a handler for VBScript
HttpRegisterHandler(hServer,
                    VIRTUAL_HOST_DEFAULT,
                    _T("vbs"),
                    _T("%SystemRoot%\\System32\\cscript.exe"),
                    _T("/nologo /b \"%1\""),
                    NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetProgramExitCode, HttpGetProgramOutput, HttpRegisterProgram

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRegisterProgram Function

```
BOOL WINAPI HttpRegisterProgram(
   HSERVER hServer,
   UINT nHostId,
   LPCTSTR lpszVirtualPath,
   LPCTSTR lpszProgramFile,
   LPCTSTR lpszParameters,
   LPCTSTR lpszDirectory
);
```

Register a CGI program for use and associate it with a virtual path on the server.

Parameters

hServer

The server handle.

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszVirtualPath

A null-terminated string which specifies the virtual path to the CGI program. This must be an absolute path, but does not have to specify a pre-existing virtual path or map to the directory structure of the root document directory for the server. This parameter cannot be NULL. The maximum length of the virtual path is 1024 characters.

lpszProgramFile

A null-terminated string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A null-terminated string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A null-terminated string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRegisterProgram** function registers an executable CGI program and associates it with a virtual path. When the client issues a GET or POST command specifying the virtual path associated with the program, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the

ability to directly access the executable file itself. It is permitted to have multiple virtual paths that reference the same executable file. The only requirement is that the virtual path be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the virtual path associated with the program. If *lpszParameters* is NULL or a zero-length string, then no additional parameters are passed to the program.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **HttpGetProgramOutput** function from within a HTTP_CLIENT_EXECUTE event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
HttpRegisterProgram(hServer,
                    VIRTUAL_HOST_DEFAULT,
                    HTTP_METHOD_DEFAULT,
                    _T("/order/invoice"),
                    _T("%ProgramData%\\MyServer\\Programs\\invoice.exe"),
                    NULL,
                    NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetProgramExitCode, HttpGetProgramOutput, HttpRegisterHandler

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRenameServerLogFile Function

```
BOOL WINAPI HttpRenameServerLogFile(
    HSERVER hServer,
    LPCTSTR lpszFileName
);
```

Rename or delete the current log file being updated by the server.

Parameters

hServer

The server handle.

lpszFileName

A null-terminated string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero. If logging is not currently enabled for the server, this function will return zero.

Remarks

The **HttpRenameServerLogFile** function is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the ***lpszFileName*** parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This function must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **HttpGetServerLogFile** function to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** functions with that file.

To disable logging, use the **HttpSetServerLogFile** function and specify the logging format as HTTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerLogFile, HttpSetServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpRequireAuthentication Function

```
BOOL WINAPI HttpRequireAuthentication(
    HSERVER hServer,
    UINT nClientId,
    UINT nAuthType,
    LPCTSTR lpszRealm
);
```

Send a response to the client indicating that authentication is required.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nAuthType

An integer value that corresponds to a result code, informing the client if the redirection is permanent or temporary. The following values may be used:

Value	Description
HTTP_AUTH_BASIC	This option specifies the Basic authentication scheme should be used. This option is supported by all clients that support at least version 1.0 of the protocol.

lpszRealm

A null-terminated string that is displayed by a web browser to indicate to the user which username and password they should use. If this parameter is NULL or an empty string, the domain name the client used to establish the connection will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRequireAuthentication** function can be used within a HTTP_CLIENT_COMMAND event handler to indicate to the client that it must provide a username and password to access the requested resource. The client should respond by issuing another request that includes the required credentials. To determine if a client has included credentials with its request, use the **HttpIsClientAuthenticated** function. The **HttpGetClientCredentials** function will return the username and password that was provided by the client.

Some clients may require that the session be secure if authentication is requested or display warning messages to the user if the connection is not secure. It is recommended that you enable security using the HTTP_OPTION_SECURE option if your application will require clients to authenticate before accessing specific resources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientCredentials, HttpIsClientAuthenticated, HttpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSendErrorResponse Function

```
BOOL WINAPI HttpSendErrorResponse(
    HSERVER hServer,
    UINT nClientId,
    UINT nErrorCode,
    LPCTSTR lpszMessage
);
```

Send a customized error response to the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nErrorCode

An integer value that specifies the error code that should be sent to the client. This value should correspond to the error result codes defined for HTTP in RFC 2616, which are three-digit values in the range of 400 through 599. The function will fail if an invalid error code is specified.

lpszMessage

A null-terminated string that describes the error. If this parameter is NULL or specifies a zero-length string, a default message will be selected based on the error code.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSendErrorResponse** function sends a response to the client indicating that an error has occurred, providing a numeric error code and HTML formatted text which may be displayed to the user. The *lpszMessage* parameter should provide a brief description of the error that will be included in the output sent to the client. Note that the message should not contain any special formatting control characters or HTML markup.

This function provides a simplified interface for sending an error response to the client. In some cases, a browser may choose to display its own error message to the user in place of the generic HTML document generated by this function. If you want your application to send a customized HTML document for a specific type of error, you should use the **HttpSendResponse** function.

If you wish to redirect the client to use an alternate URL to access the requested resource, it is recommended that you use the **HttpRedirectRequest** function rather than sending an error response.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpRedirectRequest, HttpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSendResponse Function

```
BOOL WINAPI HttpSendResponse(
  HSERVER hServer,
  UINT nClientId,
  DWORD dwOptions,
  LPHTTPRESPONSE lpResponse,
  LPVOID lpvBuffer,
  DWORD dwLength
);
```

Send a response from the server to the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be copied. It may be one of the following values:

Value	Description
HTTP_RESPONSE_DEFAULT	The ***lpvBuffer*** parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the ***dwLength*** parameter. This is the same as specifying the HTTP_RESPONSE_MEMORY option.
HTTP_RESPONSE_MEMORY	The ***lpvBuffer*** parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the ***dwLength*** parameter. The data will be sent to the client as a stream of bytes. If the server application was compiled using Unicode, it is responsibility of the application to convert any Unicode text to either ANSI or UTF-8, depending on the resource that was requested by the client.
HTTP_RESPONSE_STRING	The ***lpvBuffer*** parameter is a pointer to a string buffer that contains the response data. The maximum number of bytes of data that will be sent to the client is determined by the ***dwLength*** parameter. If the value of the ***dwLength*** parameter exceeds the string length, the value will be ignored and the contents of the string will be sent up to the terminating null character. If the Unicode version of this function is called, the string will be converted to a byte array before being sent to the client.
HTTP_RESPONSE_HGLOBAL	The ***lpvBuffer*** parameter is an HGLOBAL memory handle which references a block of memory that contains the response data. The number of bytes of data that in the

	buffer is specified by the ***dwLength*** parameter. The data will be sent to the client as a stream of bytes. It is the responsibility of the application to free the global memory handle after it is no longer needed.
HTTP_RESPONSE_FILE	The ***lpvBuffer*** parameter is a pointer to a string which specifies the name of a file that contains the response data. If the file does not exist, or does not specify a regular file, this function will fail. The ***dwLength*** parameter is ignored. If the content type for the specified file is not explicitly defined in the response, the function will attempt to automatically determine the correct type based on the file name extension and/or the contents of the file.
HTTP_RESPONSE_HANDLE	The ***lpvBuffer*** parameter is a handle to an open file and the ***dwLength*** parameter specifies the number of bytes to be read from the file and send to the client. If this option is specified, the response data will be read from the current position in the file and will advance the file pointer by the number of bytes sent to the client.
HTTP_RESPONSE_DYNAMIC	The response data will be generated dynamically. This prevents the content length from being included in the response header, and forces the connection to close, regardless if the client has requested to keep the connection open. This option must be specified if the application wishes to use the **HttpSendResponseData** function to send additional data to the client.
HTTP_RESPONSE_NOCACHE	Informs the client that the data being returned by the server should not be cached. Typically this is used in conjunction with the HTTP_RESPONSE_DYNAMIC option when the data is being generated dynamically.

lpResponse

A pointer to a **HTTPRESPONSE** structure which contains additional information about the response to the client. The structure that is passed by reference to this function must have the ***dwSize*** member initialized to the size of the structure or the function will fail. This parameter may be NULL, in which case a default response of "200 OK" is sent to the client along with any data specified by the ***lpvBuffer*** parameter.

lpvBuffer

The buffer that will contain any response data that should be sent to the client. The ***dwOptions*** parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. This parameter may be NULL, in which case no data will be sent to the client. If this function is called in response to a HEAD command being sent by the client, this parameter is ignored.

dwLength

An unsigned integer that specifies the number of bytes of data to be sent to the client. If the ***lpvBuffer*** parameter is NULL, this value must be zero. If this function is called in response to a HEAD command being sent by the client, this parameter is ignored.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSendResponse** function is called within a HTTP_CLIENT_COMMAND event handler to respond to the request made by the client. This function may only be called once after a command has been received, and must be called after the **HttpReceiveRequest** function. It is only necessary for the application to call this function if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

If the HTTP_RESPONSE_HANDLE option is used to read a copy of the response data from an open file, the handle must reference a disk file that was opened using the **CreateFile** function with GENERIC_READ access. It cannot be a handle to a device or named pipe. If the ***dwLength*** parameter is larger than the total number of bytes available to be read from the current position in the file, the function will stop sending data to the client when it reaches the end-of-file. If the function succeeds, the file pointer is advanced by the number of bytes of response data sent to the client. If the function fails, the file pointer is returned to its original position prior to the function being called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpRecieveRequest, HttpSendResponseData, HTTPRESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSendResponseData Function

```
INT WINAPI HttpSendResponseData(
   HSERVER hServer,
   UINT nClientId,
   LPBYTE lpBuffer,
   DWORD dwLength,
   LPDWORD lpdwResponse
);
```

Send additional data to the client in response to a command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A buffer that contains the response data that should be sent to the client. This parameter cannot be NULL and must point to a valid byte buffer. If the response contains non-ASCII text characters, it should be UTF-8 encoded.

dwLength

An unsigned integer that specifies the size of the response data sent to the client. This value must be greater than zero and should never exceed the number of bytes stored in the buffer specified by the *lpBuffer* parameter.

lpdwResponse

A pointer to an unsigned integer which will contain the total number of bytes successfully sent to the client in response to the request. This parameter may be NULL, in which case it is ignored.

Return Value

If the function succeeds, the return value is the number of bytes of data sent to the client. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSendResponseData** function is called within a HTTP_CLIENT_COMMAND event handler to send data to the client in response to a request. This function can only be used to send dynamically generated content after the **HttpSendResponse** function has been called. The HTTP_RESPONSE_DYNAMIC option must have been specified when responding to the client or this function will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpRecieveRequest, HttpSendResponse, HTTPRESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerAsyncNotify Function

```
BOOL WINAPI HttpServerAsyncNotify(
    HSERVER hServer,
    HWND hWnd,
    UINT uMsg
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hServer

The server handle.

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **HttpGetServerError** function.

Remarks

The **HttpServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Value	Description
HTTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the **HttpServerStart** function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The *lParam* message parameter will specify the handle to the server.
HTTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the

	application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the **HttpServerResume** function to resume accepting client connections. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the **HttpServerSuspend** function. This notification message may be sent to the application multiple times over the lifetime of the server. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the **HttpServerRestart** function. Note that the server socket handle provided by the ***lParam*** message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The ***lParam*** message parameter will specify the handle to the server.
HTTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The ***lParam*** message parameter will specify the unique ID of the client that connected to the server.
HTTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the **HttpServerStop** function. The ***lParam*** message parameter will specify the unique ID of the client that disconnected from the server.
HTTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in ***lParam*** will reference the defunct server and cannot be used with other server functions. The ***lParam*** message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The HTTP_NOTIFY_CONNECT and HTTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the ***lParam*** message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **HttpGetClientServer** function to obtain a handle to the server that created the client

session. Note that at the time the application processes the HTTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetClientServer, HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerDisableTrace Function

```
BOOL WINAPI HttpServerDisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerEnableTrace Function

```
BOOL WINAPI HttpServerEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string that specifies the name of the log file. If this parameter is NULL or points to an empty string, a log file is created in the temporary directory for the current user.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_DEFAULT	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call HttpGetServerError.

Remarks

When trace logging is enabled, the log file is opened, appended to and closed for each socket function call. Using the same file name, you can do the same in your application to add additional information to the file if needed. This can provide an application-level context for the entries made by the library. Make sure that the file is closed after the data has been written. If a file name is not specified by the caller, a file named **cstrace.log** will be created in the temporary directory for the current user.

The TRACE_HEXDUMP option can produce very large files, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpServerDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerInitialize Function

```
BOOL WINAPI HttpServerInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **HttpServerInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **HttpServerInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **HttpServerUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **HttpServerInitialize** or **HttpServerUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpServerStart, HttpServerStop, HttpServerUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerProc Function

```
VOID CALLBACK HttpServerProc(
   HSERVER hServer,
   UINT nClientId,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **HttpServerProc** function is an application-defined callback function that processes events generated by the server process.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see Server Event Constants.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero. The **HttpGetServerError** function can be used to obtain additional information about the error code.

dwParam

A user-defined integer value which was specified when the server was started. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

Return Value

None.

Remarks

The HTTP_CLIENT_COMMAND event can be used to filter commands issued by the client. If the command performs an operation on a file or directory, the **HttpGetCommandFile** function can be used to obtain the file name on the server that the client specified. It is possible for the application to change the file name using the **HttpSetCommandFile** function to direct the server to use a different file than the one specified by the client. If the file name is changed and the event handler returns a value of zero, the server will perform the default action for the command using the new file name.

The callback function will be called in the context of the thread that is currently managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within an event handler. Controls should only be modified by the same UI thread that created their window. To change the user interface in response to a server event, use the

HttpServerAsyncNotify function to enable asynchronous notifications and update the UI in response to the notification message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerAsyncNotify, HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerRestart Function

```
HSERVER WINAPI HttpServerRestart(
    HSERVER hServer
);
```

Restart the server, terminating all active client sessions.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the new handle for the specified server. If the function fails, the return value is INVALID_SERVER. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerRestart** function will restart the specified server, terminating all active client sessions. The server handle that is returned by the function is the handle for the new server instance, and the old handle value is no longer valid. If the function is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **HttpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerStart, HttpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerResume Function

```
BOOL WINAPI HttpServerResume(
    HSERVER hServer
);
```

Resume accepting client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerResume** function instructs the server to resume accepting new client connections after the **HttpServerSuspend** function has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerRestart, HttpServerStart, HttpServerStop, HttpServerSuspend, HttpServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerStart Function

```
HSERVER WINAPI HttpServerStart(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   DWORD dwOptions,
   LPHTTPSERVERCONFIG lpServerConfig,
   HTTPSERVPROC lpServerProc,
   DWORD_PTR dwServerParam,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **HttpServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to accept client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit TLS. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies the options used when creating an instance of the server. For a list of options, see Server Option Constants.

lpServerConfig

A pointer to an **HTTPSERVERCONFIG** structure that specifies the configuration options for the server. If this parameter is NULL, then default configuration parameters will be used that starts the server in a restrictive mode.

lpServerProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **HttpServerProc** callback function. If this parameter is NULL, a default internal handler is used to process client commands.

dwServerParam

A user-defined integer value that is passed to the callback function. If the ***lpServerProc*** parameter is NULL, this value should be zero. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

lpCredentials

Pointer to a **SECURITYCREDENTIALS** structure. If this parameter is NULL, the default security credentials for the server host name will be used. If security is enabled for the server, it is

recommended that you provide a pointer to this structure with specific information about the server certificate that should be used. If the security options are not specified in the server configuration, this parameter is ignored.

Return Value

If the function succeeds, the return value is a handle to a server session. If the function fails, the return value is INVALID_SERVER. To get extended error information, call **HttpGetServerError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **HttpServerStop** function.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

If the server is started with security enabled, the *lpCredentials* parameter should be used to provide the server certificate information in a **SECURITYCREDENTIALS** structure. If the pointer is NULL, this function will search for a certificate that matches the hostname provided as part of the server configuration. This requires that the server certificate be installed in the personal certificate store of the current process user, and it must have a private key associated with it. To use a certificate with a different name, or one that is stored in a PFX file, a **SECURITYCREDENTIALS** structure must be defined and passed to this function.

Example

```
HSERVER hServer;
HTTPSERVERCONFIG httpConfig;

// Initialize the server configuration
ZeroMemory(&httpConfig, sizeof(HTTPSERVERCONFIG));
httpConfig.dwSize = sizeof(httpConfig);
httpConfig.nMaxClients = 100;
httpConfig.nMaxClientsPerAdress = 8;
httpConfig.nMaxRequests = 5;
httpConfig.nLogFormat = HTTP_LOGFILE_EXTENDED;
```

```
httpConfig.nLogLevel = 5;
httpConfig.lpszIdentity = _T("MyProgram");
httpConfig.lpszHostName = _T("server.company.com");
httpConfig.lpszHostUuid = _T("10000000-1000-1000-1000-100000000000");
httpConfig.lpszDirectory = _T("%ProgramData%\\MyProgram\\Files");
httpConfig.lpszLogFile = _T("%ProgramData%\\MyProgram\\Server.log");

// Start the server
hServer = HttpServerStart(lpszLocalHost,
                          HTTP_PORT_DEFAULT,
                          HTTP_SERVER_DEFAULT,
                          &httpConfig,
                          lpEventHandler,
                          0,
                          NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpEnumServerClients, HttpServerProc, HttpServerRestart, HttpServerStop, HTTPSERVERCONFIG, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerStop Function

```
BOOL WINAPI HttpServerStop(
   HSERVER hServer
);
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **HttpServerStop** function instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **HttpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerRestart, HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerSuspend Function

```
BOOL WINAPI HttpServerSuspend(
    HSERVER hServer
);
```

Suspend the server and reject new client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **HttpServerResume** function. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **HttpServerThrottle** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerRestart, HttpServerResume, HttpServerStart, HttpServerStop, HttpServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerThrottle Function

```
BOOL WINAPI HttpServerThrottle(
  HSERVER hServer,
  UINT nMaxClients,
  UINT nMaxClientsPerAddress,
  DWORD dwConnectionRate
);
```

The **HttpServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server.

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **HttpServerStart** function is called with the maximum number of clients set to 100, and then **HttpServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would

limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerRestart, HttpServerResume, HttpServerStart, HttpServerSuspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpServerUninitialize Function

```
VOID WINAPI HttpServerUninitialize();
```

The **HttpServerUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **HttpServerInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **HttpServerUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **HttpServerUninitialize** for every successful call to **HttpServerInitialize** made by a process. Operations for all threads in the server are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpServerInitialize, HttpServerStart, HttpServerStop

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetClientAccess Function

```
BOOL WINAPI HttpSetClientAccess(
    HSERVER hServer,
    UINT nClientId,
    DWORD dwUserAccess
);
```

Change the access rights associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see User and File Access Constants.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **HttpSetClientAccess** function can change the multiple access rights for a client session. The **HttpEnableClientAccess** function can be used to grant or revoke a specific permission for the client session.

If the *dwUserAccess* parameter has a value of HTTP_ACCESS_DEFAULT, then default permissions will be granted to the client session based on the configuration of the server. This is the recommended value for most clients. It is important to consider the implications of changing the access permissions granted to a client session. For example, if you do not grant clients HTTP_ACCESS_READ permission, it can effectively disable the site because the server will return 403 Forbidden errors for all GET and HEAD requests.

This function should typically be called in the HTTP_CLIENT_CONNECT event handler to assign general permissions to the client, or in the HTTP_CLIENT_COMMAND event handler after the client has issued a request and provided any authentication credentials that are required.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpAuthenticateClient, HttpEnableClientAccess, HttpGetClientAccess

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetClientHeader Function

```
BOOL WINAPI HttpSetClientHeader(
   HSERVER hServer,
   UINT nClientId,
   UINT nHeaderType,
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue
);
```

Create or change the value of a request or response header for the client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to create or modify. It may be one of the following values:

Value	Description
HTTP_HEADERS_REQUEST	Change a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Change a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A null-terminated string that specifies the name of the header that should be created or modified. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A null-terminated string that specifies the new value of the header. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the header value from the list of request or response headers.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpSetClientHeader** function will change the value of a request or response header for the specified client session. If the ***lpszHeaderName*** value matches an existing header field, its value will be replaced. If the header name is not defined, then a new header will be created with the given value.

You should not change the value of most standard response header values unless you are certain of the impact that it would have on the normal operation of the client.

If you wish to define a custom header value that would be included in the response to a client request, you should prefix the header name with "X-" to avoid potential conflicts with the standard response headers. For example, if you wanted to identify a customer, you could create a header field with the name "X-Customer-ID" and set the value to the customer ID number. The client application would receive this custom header value as part of the response to its request for a document.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpDeleteClientHeader, HttpGetClientHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetClientIdleTime Function

```
UINT WINAPI HttpSetClientIdleTime(
    HSERVER hServer,
    UINT nClientId,
    UINT nTimeout
);
```

Change the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the function succeeds, the return value is the previous client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpSetClientIdleTime** function is will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **HttpGetClientIdleTime** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetClientIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetClientVariable Function

```
BOOL WINAPI HttpSetClientVariable(
    HSERVER hServer,
    UINT nClientId,
    LPCTSTR lpszName,
    LPCTSTR lpszValue
);
```

Create or change the value of a CGI environment variable for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A null-terminated string that specifies the name of the environment variable that should be created or modified. Environment variables are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A null-terminated string that specifies the new value of the environment variable. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the variable from the environment block for the client session.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpSetClientVariable** function will change the value of a environment variable for the specified client session. If the ***lpszName*** value matches an existing variable, its value will be replaced. If the variable is not defined, then a new variable will be created with the given value. The value of an environment variable can be obtained using the **HttpGetClientVariable** function.

The server will automatically create a number of different environment variables that will be passed to a program or script executed by the server. These variables are defined in RFC 3875 as part of the Common Gateway Interface (CGI) 1.1 specification. The following variables are defined by the server and should not be modified directly by the application:

Variable Name	Description
AUTH_TYPE	The authorization scheme used by the server to authenticate the client session
CONTENT_LENGTH	The length of the request data provided by the client
CONTENT_TYPE	The MIME type that identifies the type of content provided by the client
DOCUMENT_ROOT	The full path to the local document root directory on the server

GATEWAY_INTERFACE	The version of the Common Gateway Interface that is being used by the server
PATH_INFO	The resource or sub-resource that is to be returned by the program or script
PATH_TRANSLATED	The path information mapped to the server root document directory structure
QUERY_STRING	The URL encoded query parameters passed to the program or script
REMOTE_ADDR	The network address of the client sending the request to the server
REMOTE_HOST	The same value as the REMOTE_ADDR variable
REMOTE_USER	The username specified as part of the authentication credentials provided by the client
REQUEST_METHOD	The method used by the client to request the resource
REQUEST_URI	The URI for the script provided by the client
SCRIPT_FILENAME	The full path to the program or script on the server
SCRIPT_NAME	The path to the program or script specified by the client
SERVER_NAME	The hostname or IP address of the server that the client connected to
SERVER_PORT	The port number that the client used to connect to the server
SERVER_PORT_SECURE	This variable has a value of "1" if the client connection to the server is secure
SERVER_PROTOCOL	The version of the server protocol used
SERVER_SOFTWARE	The server identity string which specifies the application name and version

In addition to the environment variables listed, the server will also create variables that are prefixed with "HTTP_" that are set to the value of request headers that are not otherwise defined. For example, the HTTP_USER_AGENT variable will be set to the value of the User-Agent header provided by the client as part of the request.

Calling the **HttpSetClientVariable** function in response to an HTTP_CLIENT_EXECUTE event notification will have no effect because it occurs after the CGI program or script has completed execution. To create or modify environment variables for the client session, it should be done in response to the HTTP_CLIENT_COMMAND event notification.

This function will not change the environment block for the server process. Each client session is allocated its own private environment block which is inherited by the CGI program. When the client session terminates, the memory allocated for its environment is released.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientVariable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetCommandFile Function

```
BOOL WINAPI HttpSetCommandFile(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszFileName
);
```

Change the name of the local file or directory that is the target of the current command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A null-terminated string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpSetCommandFile** function is used by the application to change the target file or directory name for the current command from within an HTTP_EVENT_COMMAND event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the GET command to download a file from the server, this function can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **HttpGetCommandFile** function.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the root directory for the client session. Because the root document directory for the client can depend on the hostname that the client used when connecting to the server, it is recommended that you always use an absolute path. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this function to change the target file name.

Typically this function would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this function to provide access to data that is stored outside of the server root directory. Incorrect use of this function could expose the server to security risks or cause unpredictable behavior by client applications. In most cases it is preferable to use the **HttpAddVirtualPath** function to create a virtual path or file name on the server, or the **HttpRedirectRequest** function to request the client use a different URL to access the resource.

This function should only be called within the context of the HTTP_EVENT_COMMAND event, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this function will return zero and the last error code will be set to

ST_ERROR_INVALID_COMMAND.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpAddVirtualPath, HttpGetCommandFile, HttpReceiveRequest, HttpRedirectRequest, HttpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetServerError Function

```
VOID WINAPI HttpSetServerError(
    HSERVER hServer,
    DWORD dwError
);
```

Set the last error code for the specified server session.

Parameters

hServer

The server handle.

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR.

If the ***hServer*** parameter is specified as INVALID_SERVER, this function will set the last error code for the current thread, but will not change the error code associated with any server session. This should only be done if the application does not have access to a valid server handle.

If the ***dwError*** parameter is specified with a value of zero, this effectively clears the error code for the last function that failed. Those functions which clear the last error code when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **HttpGetServerError** function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetCommandResult, HttpGetServerError, HttpSendResponse

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetServerIdentity Function

```
BOOL WINAPI HttpSetServerIdentity(
    HSERVER hServer,
    LPCTSTR lpszIdentity
);
```

Change the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A null-terminated string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **HttpSetClientIdentity** function changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **HttpGetServerIdentity** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerIdentity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetServerLogFile Function

```
BOOL WINAPI HttpSetServerLogFile(
   HSERVER hServer,
   UINT nLogFormat,
   UINT nLogLevel,
   LPCTSTR lpszFileName
);
```

Change the current log format, level of detail and file name.

Parameters

hServer

The server handle.

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Value	Description
HTTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about GET, PUT and POST requests.
HTTP_LOGFILE_COMBINED	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session. This format typically generates the largest logfiles.

nLogLevel

An integer value that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of HTTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A null-terminated string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero.

Remarks

The **HttpSetServerLogFile** function can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the ***nLogFormat*** parameter with the constant HTTP_LOGFILE_DELETE.

The higher the value of the ***nLogLevel*** parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record GET, POST and PUT requests, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = HTTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (HttpGetServerLogFile(hServer, &nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != HTTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = HttpSetServerLogFile(hServer, nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerLogFile, HttpRenameServerLogFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetServerName Function

```
BOOL WINAPI HttpSetServerName(
   HSERVER hServer,
   UINT nClientId,
   LPCTSTR lpszHostName
);
```

Change the host name assigned to the specified server or client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A null-terminated string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will change the host name assigned to the specified client session. If the ***nClientId*** parameter has a value of zero, the function will change default host name that was assigned to the server as part of the server configuration. If the ***nClientId*** parameter specifies a valid client session and the ***lpszHostName*** parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **HttpGetServerName** function will return the host name specified by the client, and **HttpSetServerName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerAddress, HttpGetServerName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetServerPriority Function

```
INT WINAPI HttpSetServerPriority(
    HSERVER hServer,
    INT nPriority
);
```

Change the priority assigned to the specified server.

Parameters

hServer

The server handle.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Value	Description
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetServerPriority, HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HttpSetServerStackSize Function

```
BOOL WINAPI HttpSetServerStackSize(
    SOCKET hServer,
    DWORD dwStackSize
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtsv12.lib

See Also

HttpGetServerMemoryUsage, HttpGetServerStackSize, HttpServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Hypertext Transfer Protocol Server Data Structures

- HTTPCLIENTCREDENTIALS
- HTTPREQUEST
- HTTPRESPONSE
- HTTPSERVERCONFIG
- HTTPSERVERTRANSFER
- SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPCLIENTCREDENTIALS

The **HTTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _HTTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    UINT  nAuthType;
    UINT  nHostPort;
    TCHAR szHostName[HTTP_MAXHOSTNAME];
    TCHAR szUserName[HTTP_MAXUSERNAME];
    TCHAR szPassword[HTTP_MAXPASSWORD];
} HTTPCLIENTCREDENTIALS, *LPHTTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value is reserved for future use. This value will always be zero.

nAuthType

An unsigned integer value that identifies the authentication method.

nHostPort

The server port number.

szHostName

A null-terminated string that specifies the server host name.

szUserName

A null-terminated string that specifies the user name.

szPassword

A null-terminated string that specifies the user password.

Remarks

When an instance of this structure is passed to the **HttpGetClientCredentials** function, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetClientCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPREQUEST

The **HTTPREQUEST** structure provides information about the request made by the client.

```
typedef struct _HTTPREQUEST
{
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwVersion;
    DWORD  dwAccess;
    DWORD  dwLength;
    DWORD  dwReserved;
    UINT   nHostId;
    UINT   nHostPort;
    LPCSTR lpszCommand;
    LPCSTR lpszHostName;
    LPCSTR lpszUserName;
    LPCSTR lpszPassword;
    LPCSTR lpszResource;
    LPCSTR lpszParameters;
    LPCSTR lpszDirectory;
    LPCSTR lpszPathInfo;
    LPCSTR lpszProgram;
    LPCSTR lpszScriptFile;
    LPCSTR lpszLocalFile;
    LPCSTR lpszMediaType;
} HTTPREQUEST, *LPHTTPREQUEST;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that provides additional information about the request. It may be one or more of the following values:

Value	Description
HTTP_REQUEST_FLAG_PROTECTED	The resource that the client has requested is protected. The client should only be permitted to access the resource if the client session has been authenticated.
HTTP_REQUEST_FLAG_PROGRAM	The resource that the client has requested is a CGI program or an executable script. The output from the program or script should be returned to the client in the response. If this flag is specified, the ***lpszProgram*** member of the structure specifies the name of the program that should be executed.
HTTP_REQUEST_FLAG_SCRIPT	The resource that the client has requested is an executable script. This flag is set when the request URL is mapped to a registered script handler. The ***lpszProgram*** member specifies the name of the program that is responsible for executing the script, and the ***lpszScriptFile*** parameter specifies the full

	path to the script itself.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Value	Description
HTTP_VERSION_09	The client issued a GET command without specifying a protocol version.
HTTP_VERISON_10	The client issued a command that specified version 1.0 of the protocol.
HTTP_VERSION_11	The client issued a command that specified version 1.1 of the protocol.

dwAccess

An unsigned integer value that specifies the access permissions that were assigned to the resource. For a list of file access permissions, see User and File Access Constants.

dwLength

An unsigned integer value that will specify the number of bytes of request data provided by the client. If the client did not submit any data with the request, this member will have a value of zero.

dwReserved

An unsigned integer value that is reserved for future use.

nHostId

An integer value that identifies the virtual host that was specified by the client. This is based on the value of the Host request header included in the request. This value will be zero if a host name is not specified by the client, or does not match one of the virtual hosts that have been created.

nHostPort

An integer value that specifies the port number that the client used to establish the connection. This will be the same port number that was used when starting the server.

lpszCommand

A null-terminated string that specifies the command that was issued by the client. The command will always be in upper case. Refer to Hypertext Transfer Protocol Commands for a list of standard commands.

lpszHostName

A null-terminated string that identifies the hostname or IP address that the client used to establish the connection. This will typically correspond to the hostname assigned to the server, or to one of the virtual hosts that have been created.

lpszUserName

A null-terminated string that specifies the username provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszPassword

A null-terminated string that specifies the password provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszResource

A null-terminated string that specifies the URL path provided by the client.

lpszParameters

A null-terminated string that specifies any query parameters that were included in the request made by the client. If the URL did not include any query parameters, this member will be NULL.

lpszDirectory

A null-terminated string the specifies the full path to the root directory for the virtual host.

lpszPathInfo

A null-terminated string that specifies additional path information provided by the client when referencing an executable CGI program or script in the URL. For example, if a program is mapped to the virtual path "/orders/invoice" and the client requests "/orders/invoice/pdf/10001" this member will be the string "/pdf/10001". If there is no path information associated with the request, this member will be NULL.

lpszProgram

A null-terminated string that specifies the local path to the CGI program that should be executed to process the request made by the client. If this member is NULL, it indicates that the client has provided a URL that maps to a local file or directory.

lpszScriptFile

A null-terminated string that specifies the local path to the script file that will be executed by the handler. If this member is NULL, it indicates that the request URL was not mapped to registered script handler.

lpszLocalFile

A null-terminated string that specifies the local path to the directory or file name referenced by the URL. If the ***lpszPathInfo*** member is not NULL, then this member will point to a local file name based the path information appended to the root directory for the virtual host.

lpszMediaType

A null-terminated string that identifies the request data using the standard Internet media types defined in RFC 2046. It is used to designate the format for various types of content and has two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If the client did not submit any data with the request, this member will be NULL.

Remarks

This structure is used with the **HttpReceiveRequest** function and all members should be initialized to a value of zero, except for the ***dwSize*** member which should be initialized to size of the structure. Failure to properly initialize the structure will cause the function call to fail.

The value of the ***lpszLocalFile*** member depends on whether the client has requested a static document, or if the URL is mapped to a registered program or script handler. If the request is for a static document, then the ***dwFlags*** member will not have the HTTP_REQUEST_FLAG_PROGRAM bit flag set and the ***lpszLocalFile*** member will be the full path to the document. If the URL is mapped to a registered program or script, then the ***lpszLocalFile*** member will be the full path to the server root directory, If the request URL included additional path information, that will be appended to the root directory.

The value of the ***lpszScriptFile*** member is only defined if the ***dwFlags*** member has the HTTP_REQUEST_FLAG_SCRIPT bit flag set. This indicates that the request URL references a file that is an executable script with a handler that was registered using the **HttpRegisterHandler** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpReceiveRequest, HttpSendResponse, HTTPRESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPRESPONSE

The **HTTPRESPONSE** structure provides additional information about the response to the client.

```
typedef struct _HTTPRESPONSE
{
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwVersion;
    DWORD   dwReserved;
    UINT    nExpires;
    UINT    nResultCode;
    LPCTSTR lpszReason;
    LPCTSTR lpszMediaType;
} HTTPRESPONSE, *LPHTTPRESPONSE;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that specifies one or more response flags. This member is reserved for future use and should always have a value of zero.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Value	Description
HTTP_VERSION_DEFAULT	The server should respond to the client using the same version specified in the request.
HTTP_VERISON_10	The server should respond to the client using version 1.0 of the protocol.
HTTP_VERSION_11	The server should respond to the client using version 1.1 of the protocol.

dwReserved

An unsigned integer value that is reserved for future use.

nExpires

An integer value that specifies the number of seconds until the client should consider any cached response data to be stale. If this value is zero, the default cache expiration time for the server will be used. The default cache expiration time is 7200 seconds (2 hours). The value of this structure member is ignored if the HTTP_RESPONSE_NOCACHE option is specified when the response is sent to the client.

nResultCode

An integer value that specifies the result code that should be sent in response to the client request. The result code is a three-digit number that indicates success or failure. For more information, refer to the **HttpGetCommandResult** function.

lpszReason

A string that that describes the result code sent to the client. The description should be brief and

should not contain any formatting characters or HTML markup. This parameter may be NULL, in which case a default description of the result code will be used.

lpszMediaType

A string that specifies the Internet media type for the data that is being sent to the client as defined in RFC 2046. The format for the content type string consists of two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If this member is NULL, the library will attempt to automatically determine the appropriate media type.

Remarks

This structure is used with the **HttpSendResponse** function and the *dwSize* member must be initialized to size of the structure. Failure to properly initialize the structure will cause the function call to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetCommandResult, HttpReceiveRequest, HttpSendResponse, HTTPREQUEST

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPSERVERCONFIG

The **HTTPSERVERCONFIG** structure provides the configuration information used to create an instance of an HTTP server.

```
typedef struct _HTTPSERVERCONFIG
{
    DWORD   dwSize;
    UINT    nMaxClients;
    UINT    nMaxClientsPerAddress;
    UINT    nMaxRequests;
    UINT    nMaxPostSize;
    UINT    nLogFormat;
    UINT    nLogLevel;
    UINT    nCacheTime;
    UINT    nIdleTime;
    UINT    nExecTime;
    LPCTSTR lpszIdentity;
    LPCTSTR lpszHostUuid;
    LPCTSTR lpszHostName;
    LPCTSTR lpszRootPath;
    LPCTSTR lpszTempPath;
    LPCTSTR lpszLogFile;
} HTTPSERVERCONFIG, *LPHTTPSERVERCONFIG;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

nMaxClients

An integer value that specifies the maximum number of active client connections that will be accepted by the server. If the maximum number of clients is reached, any further connections are rejected by the server until one or more clients close their connection to the server or are disconnected. A value of zero specifies the default configuration value of 100 connections should be used.

nMaxClientsPerAddress

An integer value that specifies the maximum number of active client connections per IP address that will be accepted by the server. If the maximum number of clients from the same IP address is reached, any further connections are rejected until one of the clients closes its connection to the server. A value of zero specifies the default configuration value of 8 connections per address should be used.

nMaxRequests

An integer value that specifies the maximum number of a requests a client may make per connection. When a client reaches the maximum number of requests, the server will close the connection and the client must establish a new connection to send another request. If this member has a value of zero, the server will allow 5 requests per connection. The minimum value is 1 and the maximum is 100.

nMaxPostSize

An integer value that specifies the maximum size of the request data that may be submitted by a client with a POST command. If the amount of request data submitted to the server exceeds this limit, the server will return a 413 error to the client and the connection will be closed. If this member

has a value of zero, the server will permit the client to submit up to 4 MB (4,194,304 bytes) of data. The minimum value is 100 KB (102,400 bytes) and the maximum value is 100 MB (104,857,600 bytes).

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Value	Description
HTTP_LOGFILE_NONE	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If the ***nLogFormat*** member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format. The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

nCacheTime

An integer value that specifies the amount of time that the client should store a document in its cache before requesting it from the server again. A value of zero specifies that the default value of 7200 seconds (2 hours) should be used.

nIdleTime

An integer value that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 10 seconds and the maximum value is 300 seconds (5 minutes). This value is used to initialize the default idle timeout period for each client session. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress.

nExecTime

An unsigned integer value that specifies the maximum number of seconds that an external CGI program is permitted to run on the server. Programs are registered using the **HttpRegisterProgram** function, and are executed when the client sends a request for a resource that is associated with the program. If this value is zero, the default timeout period of 5 seconds will

be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds.

lpszIdentity

A null-terminated string that identifies the server. It is used for informational purposes only and has no effect on the operation of the server. If this member is not initialized to a value, then a default identity will be used. The server identity is provided when a client establishes the initial connection and when the client sends the STAT or CSID commands. If this member is defined, it is recommended that you only use printable ASCII characters.

lpszHostUuid

A null-terminated string that specifies a Universally Unique Identifier (UUID) that is used to uniquely identify the server. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This structure member may be initialized to an empty string, in which case a temporary UUID will be randomly generated.

lpszHostName

A null-terminated string that specifies the fully-qualified host name for the server. If this structure member is initialized with an empty string, the local host name assigned to the Windows system will be used. This value does not need to correspond to the actual host name associated with the server's IP address. This value is used for informational purposes only and has no effect on the operation of the server.

lpszRootPath

A null-terminated string that specifies the path to the root directory for the server. If this structure member is initialized with an empty string, the current working directory for the process will be used as the root directory. It is recommended that you provide a full path to an existing directory and do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszTempPath

A null-terminated string that specifies the path to the temporary directory for the server. If this structure member is initialized with an empty string, the current temporary directory for the process will be used. The temporary directory cannot be the same as the root directory, and it is strongly recommended that you do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszLogFile

A null-terminated string that specifies the name of the server log file to create, if a logging format has been specified. If logging is enabled and this member is an empty string, then a default log file name will be created. If the file name does not include a path, then the file is created in the server log directory. If the file name includes a path, the log file will be created using that specific name. If the server is in multi-user mode, then the default location for log files will be the Logs subdirectory in the server root directory. If the server is not in multi-user mode, the default location for log files will be the temporary directory for the current process.

Remarks

When an instance of this structure is passed to the **HttpServerStart** function, the *dwSize* member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the configuration is invalid.

The *nMaxClients* member limits the total number of client connections and the *nMaxClientsPerAddress* member limits the number of simultaneous connections can originate from the same address. If either client limit is exceeded, the server will automatically close the connection.

For more control over how the server accepts client connections, use the **HttpServerThrottle** function. It is not recommended that you set the maximum clients per address below a value of 4. Lower values can negatively impact the performance of some clients, particularly web browsers.

The ***nMaxPostSize*** member limits the amount of data that a client can submit to the server using a POST command. The default limit is 4 MB which should be sufficient for most applications. One situation where you may need to increase this value is if you expect the client to upload large files using the POST command. Increasing this limit will potentially increase the amount of virtual memory that the server will allocate. Note that this value does not limit amount of data that can be uploaded using the PUT command.

When specifying the root directory for the server configuration, it is important to consider that a client's default access to files on the system will be limited to these directories and any subdirectories. If a root directory is not specified, then one will be created using the current working directory for the process. The root directory path may include environment variables surrounded by % symbols and these will be expanded.

For servers that are publicly accessible, or where you want files to be a accessible across multiple server sessions, you should always populate the ***szHostUuid*** member with a valid UUID string, and the ***lpszRootPath*** member should specify an absolute path to an existing directory.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpServerStart, HttpServerStop, HttpServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

HTTPSERVERTRANSFER

The **HTTPSERVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _HTTPSERVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} HTTPSERVERTRANSFER, *LPHTTPSERVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Value	Description
HTTP_FILE_READ	The file was opened for reading. This mode indicates that the client issued the GET command to download the contents of a file from the server to the client system. The ***szFileName*** member specifies the name of the local file on the server that was downloaded by the client.
HTTP_FILE_WRITE	The file was opened for writing. This mode indicates that the client issued the PUT command to upload the contents of a file from the client system to server. The ***szFileName*** member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a ULARGE_INTEGER union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file.

szFileName

A null-terminated string value that will contain the full path to the local file that was transferred. The ***dwFileAccess*** member determines whether the file name represents a file that was downloaded by

the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **HttpGetServerTransferInfo** function, the *dwSize* member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

HttpGetServerTransferInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify the security protocol and server certificate when accepting a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol Library

Determine if a remote host is reachable and how packets of data are routed to that system.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSICMV12.DLL
Version	12.0.3024.1286
LibID	66D5E749-831F-40CF-9B36-77A849C6C0E5
Import Library	CSICMV12.LIB
Dependencies	None
Standards	RFC 792

Overview

The Internet Control Message Protocol (ICMP) is commonly used to determine if a remote host is reachable and how packets of data are routed to that system. Users are most familiar with this protocol as it is implemented in the ping and traceroute command line utilities. The ping command is used to check if a system is reachable and the amount of time that it takes for a packet of data to make a round trip from the local system, to the remote host and then back again. The traceroute command is used to trace the route that a packet of data takes from the local system to the remote host, and can be used to identify potential problems with overall throughput and latency. The library can be used to build in this type of functionality in your own applications, giving you the ability to send and receive ICMP echo datagrams in order to perform your own analysis.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports

native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol Functions

Function	Description
IcmpCancel	Cancel the current blocking operation
IcmpCloseHandle	Close the current client session
IcmpCreateHandle	Create a new handle for the current client session
IcmpDisableEvents	Disable asynchronous event notification
IcmpDisableTrace	Disable logging of network-level calls to the trace log
IcmpEcho	Send one or more echo datagrams to the specified host
IcmpEnableEvents	Enable asynchronous event notification
IcmpEnableTrace	Enable logging of network function calls to a file
IcmpEventProc	Callback function that processes events generated by the client
IcmpFormatAddress	Convert a numeric IP address to a string
IcmpFreezeEvents	Suspend or resume event handling by the calling process
IcmpGetErrorString	Return a description for the specified error code
IcmpGetHostAddress	Return the current host IP address
IcmpGetHostName	Return the current host name
IcmpGetLastError	Return the last error code
IcmpGetPacketSize	Return the ICMP datagram packet size
IcmpGetRecvCount	Return the number of packets received
IcmpGetSendCount	Return the number of packets sent
IcmpGetSequenceId	Return the current sequence identifier
IcmpGetTimeout	Return the number of milliseconds until an operation times out
IcmpGetTimeToLive	Return the current time-to-live for the ICMP datagram
IcmpGetTripTime	Return the current trip time statistics
IcmpInitialize	Initialize the library and validates the specified license key at runtime
IcmpIsBlocking	Determine if the client is blocked, waiting for information
IcmpRecvEcho	Read an ICMP datagram returned by the remote host
IcmpRegisterEvent	Register an event callback function
IcmpReset	Reset the current client state
IcmpResolveAddress	Resolve an IP address into a fully qualified host name
IcmpSendEcho	Send an ICMP datagram to the specified host
IcmpSetHostAddress	Set the IP address of the host to receive the next datagram
IcmpSetHostName	Set the name of the host to receive the next datagram

IcmpSetLastError	Set the last error code
IcmpSetPacketSize	Set the ICMP datagram packet size
IcmpSetSequenceId	Set the sequence identifier for the next datagram
IcmpSetTimeout	Set the number of milliseconds until an operation times out
IcmpSetTimeToLive	Set the time-to-live for the next datagram
IcmpTraceRoute	Trace the route from the local system to a remote host
IcmpUninitialize	Terminate use of the library by the application

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpCancel Function

```
INT WINAPI IcmpCancel(
   HCLIENT hClient
);
```

The **IcmpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

When the **IcmpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpCloseHandle Function

```
INT WINAPI IcmpCloseHandle(
   HCLIENT hClient
);
```

The **IcmpCloseHandle** function closes the socket and releases the memory allocated for the client session.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpCreateHandle, IcmpInitialize, IcmpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpCreateHandle Function

```
HCLIENT WINAPI IcmpCreateHandle(
    UINT nPacketSize,
    UINT nTimeToLive,
    DWORD dwTimeout,
    DWORD dwReserved,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **IcmpCreateHandle** function creates a client handle for sending and receiving ICMP echo datagrams. If an event notification window is specified, the client will be notified when a network event occurs.

Parameters

nPacketSize

An unsigned integer which specifies the default packet size used when generating ICMP echo datagrams. The minimum packet size is 32 bytes and the maximum size is 65,535 bytes.

nTimeToLive

An unsigned integer which specifies the default time-to-live for ICMP echo datagrams. This determines the maximum number of times that a packet will be routed from one system to another while en route to its destination. The minimum time-to-live value is 1, the maximum is 255. The recommend value for this parameter is 255, and typical applications should use a time-to-live value of at least 30.

dwTimeout

An unsigned integer which specifies the maximum number of milliseconds to wait before the current operation times out.

dwReserved

A reserved parameter. This value should always be zero.

hEventWnd

The handle to an asynchronous notification window. This window receives messages which notify the client when asynchronous network events occur. If asynchronous event notification is not required, this parameter may be NULL.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to the client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpCreateHandle** function creates a client handle that is used with subsequent calls to the library. This library uses a special type of socket called a raw socket, which is created to send and receive ICMP echo datagrams. Raw socket support is optional under the Windows Sockets specification, and may not be available if a non-standard networking libraries are used or may only be

available to privileged accounts.

If the ***hEventWnd*** parameter is not NULL, the client operates in asynchronous mode and messages will be posted to the notification window when a network event occurs. When a message is posted to the window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. TThe client application may attempt to retry the operation or close the handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **IcmpDisableEvents** function.

The ability to create and send ICMP echo datagrams is limited to privileged users. Non-administrator users will receive an error if they attempt to create a client handle. On Windows NT it is possible to disable this security check by creating or modifying the system registry. Microsoft Knowledge Base article 195445 has additional information and instructions for making this change.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpCloseHandle, IcmpInitialize, IcmpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpDisableEvents Function

```
INT WINAPI IcmpDisableEvents(
    HCLIENT hClient
);
```

The **IcmpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpEnableEvents, IcmpFreezeEvents, IcmpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpDisableTrace Function

```
BOOL WINAPI IcmpDisableTrace();
```

The **IcmpDisableTrace** function disables the logging of network function calls file.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpEcho Function

```
INT WINAPI IcmpEcho(
   LPCTSTR lpszHostName,
   UINT nIterations,
   UINT nPacketSize,
   DWORD dwTimeout,
   DWORD dwReserved,
   LPICMPTIME lpTime
);
```

The **IcmpEcho** function sends one or more ICMP echo datagrams, collecting information about the reliability and latency of a connection between the local and remote host.

Parameters

lpszHostName

A null-terminated string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nIterations

An unsigned integer value which specifies the number of echo datagrams that will be sent to the remote host. The minimum value for this parameter is 1 and the maximum value is 512.

nPacketSize

An unsigned integer value which specifies the size of the echo datagram in bytes. The minimum size is 1 byte and the maximum size is 65,535 bytes. It is recommended that most applications use the minimum size of 32 bytes for this parameter.

dwTimeout

An unsigned integer which specifies the number of milliseconds the function will wait for a response to an echo datagram.

dwReserved

A reserved parameter. This value should always be zero.

lpTime

A pointer to an ICMPTIME structure which will contain the minimum, maximum and average round trip times for the echo datagrams sent and received.

Return Value

If the function succeeds, the return value is the number of replies received from the remote host. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpEcho** function sends a series of ICMP echo datagrams to the specified host, providing a simplified interface for pinging a remote system. If the function returns the same value as the number of iterations, then replies were received for all of the echo datagrams that were sent. This would typically indicate that the client can establish a reliable connection to the server; the values returned in the ICMPTIME structure provide information on the latency between the two hosts. Higher average time values would indicate greater latency and reduced throughput between the systems. If the function returns a value less than the specified number of iterations, this indicates that replies were not received for one or more of the echo datagrams. This may indicate that there are problems routing

data between the local and remote host. A return value of zero indicates that there was no response to the echo datagrams. The remote host may not exist or may not be available.

The failure for a host to respond to an ICMP echo datagram may not indicate a problem with the remote system. In some cases, a router between the local and remote host may be malfunctioning or discarding the datagrams. Systems can also be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpCreateHandle, IcmpGetTripTime, IcmpRecvEcho, IcmpSendEcho, IcmpTraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpEnableEvents Function

```
INT WINAPI IcmpEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **IcmpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **IcmpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. TThe client application may attempt to retry the operation or close the handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **IcmpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpDisableEvents, IcmpFreezeEvents, IcmpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpEnableTrace Function

```
BOOL WINAPI IcmpEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **IcmpEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpEventProc Function

```
VOID CALLBACK IcmpEventProc(
   HCLIENT hClient,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **IcmpEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **IcmpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **IcmpRegisterEvent** function. The **IcmpEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpDisableEvents, IcmpEnableEvents, IcmpFreezeEvents, IcmpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpFormatAddress Function

```
INT WINAPI IcmpFormatAddress(
   DWORD dwAddress,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **IcmpFormatAddress** function converts a numeric IP address to a string.

Parameters

dwAddress

A unsigned 32-bit integer which specifies the IP address to be converted into a string.

lpszAddress

A null-terminated string which will contain the converted IP address in dot-notation. This string should be at least 16 characters in length.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the function succeeds, the return value is the length of the string buffer. If the function fails, the return value is zero. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpGetHostAddress, IcmpGetHostName, IcmpResolveAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpFreezeEvents Function

```
INT WINAPI IcmpFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **IcmpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpDisableEvents, IcmpEnableEvents, IcmpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetErrorString Function

```
INT WINAPI IcmpGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **IcmpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **IcmpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpGetLastError, IcmpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetHostAddress Function

```
INT WINAPI IcmpGetHostAddress(
   HCLIENT hClient,
   LPTSTR lpszHostAddress,
   INT nMaxLength
);
```

The **IcmpGetHostAddress** copies the IP address of the current host into the specified buffer as a string using dot-notation.

Parameters

hClient

A handle to the client session.

lpszHostAddress

The buffer that will contain the IP address of the current remote host in dot-notation. This buffer should be at least 16 characters in length.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the function succeeds, the return value is the length of the address string. If the return value is zero, this indicates that no remote host has been specified. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpFormatAddress, IcmpGetHostName, IcmpResolveAddress, IcmpSetHostAddress, IcmpSetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetHostName Function

```
INT WINAPI IcmpGetHostName(
   HCLIENT hClient,
   LPTSTR lpszHostName,
   INT cbHostName
);
```

The **IcmpGetHostName** copies the name of the current host into the specified buffer.

Parameters

hClient

A handle to the client session.

lpszHostName

The buffer that will contain the name of the current remote host. This buffer should be at least 64 characters in length.

cbHostName

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the function succeeds, the return value is the length of the host name string. If the return value is zero, this indicates that no remote host has been specified. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpFormatAddress, IcmpGetHostAddress, IcmpResolveAddress, IcmpSetHostAddress, IcmpSetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetLastError Function

```
DWORD WINAPI IcmpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **IcmpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **IcmpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **IcmpSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those functions which call **IcmpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetErrorString, IcmpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetPacketSize Function

```
UINT WINAPI IcmpGetPacketSize(
    HCLIENT hClient
);
```

The **IcmpGetPacketSize** function returns the size of the ICMP echo datagram that will be sent to the remote host. The minimum packet size is 1 byte, the maximum packet size is 65,535 bytes.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the size of the datagram. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetRecvCount, IcmpGetSendCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpGetTripTime, IcmpSetPacketSize, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetRecvCount Function

```
INT WINAPI IcmpGetRecvCount(
    HCLIENT hClient
);
```

The **IcmpGetRecvCount** function returns the number of replies sent to the client from the remote host.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the number of replies. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetSendCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpGetTripTime, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetSendCount Function

```
INT WINAPI IcmpGetSendCount(
    HCLIENT hClient
);
```

The **IcmpGetSendCount** function returns the number of datagrams sent to the remote host by the client.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the number of datagrams sent by the client. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetRecvCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpGetTripTime, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetSequenceId Function

```
INT WINAPI IcmpGetSequenceId(
    HCLIENT hClient
);
```

The **IcmpGetSequenceId** function returns the sequence identifier for the last ICMP echo datagram received by the client.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the sequence identifier. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetRecvCount, IcmpGetSendCount, IcmpGetTimeToLive, IcmpGetTripTime, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetTimeout Function

```
DWORD WINAPI IcmpGetTimeout(
    HCLIENT hClient
);
```

The **IcmpGetTimeout** function returns the number of milliseconds the client will wait for a response from the remote host. Once the specified number of milliseconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in milliseconds. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpIsBlocking, IcmpSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetTimeToLive Function

```
INT WINAPI IcmpGetTimeToLive(
    HCLIENT hClient
);
```

The **IcmpGetTimeToLive** function returns the time-to-live for the last ICMP echo datagram received by the client.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the time-to-live for the last datagram. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded. The default value is 255.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetRecvCount, IcmpGetSendCount, IcmpGetSequenceId, IcmpGetTripTime, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpGetTripTime Function

```
INT WINAPI IcmpGetTripTime(
   HCLIENT hClient,
   LPICMPTIME lpIcmpTime
);
```

The **IcmpGetTripTime** returns round-trip statistics for the datagrams sent to the current remote host.

Parameters

hClient

A handle to the client session.

lpIcmpTime

A pointer to an ICMPTIME data structure which will contain the round-trip statistics for the current remote host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetRecvCount, IcmpGetSendCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpInitialize Function

```
BOOL WINAPI IcmpInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **IcmpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **IcmpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **IcmpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **IcmpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **IcmpInitialize** or **IcmpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpCloseHandle, IcmpCreateHandle, IcmpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpIsBlocking Function

```
BOOL WINAPI IcmpIsBlocking(
    HCLIENT hClient
);
```

The **IcmpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IcmpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpCancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpRecvEcho Function

```
INT WINAPI IcmpRecvEcho(
   HCLIENT hClient
);
```

The **IcmpRecvEcho** function reads the reply to an ICMP echo datagram generated by the client. This function should only be called by asynchronous client sessions in response to an event notification that a datagram has been received.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetRecvCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpSendEcho

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpRegisterEvent Function

```
INT WINAPI IcmpRegisterEvent(
   HCLIENT hClient,
   UINT nEventId,
   INETEVENTPROC lpfnEvent,
   DWORD_PTR dwParam
);
```

The **IcmpRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Value	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. The client application may attempt to retry the operation or close the handle.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **IcmpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpRegisterEvent** function associates a callback function with a specific event. The event handler is an **IcmpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will

be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpDisableEvents, IcmpEnableEvents, IcmpEventProc, IcmpFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpReset Function

```
INT WINAPI IcmpReset(
   HCLIENT hClient
);
```

The **IcmpReset** function resets the client session, clearing the packet trip statistics, time-to-live, sequence identifier, send and receive counts.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpSetHostAddress, IcmpSetHostName, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpResolveAddress Function

```
INT WINAPI IcmpResolveAddress(
   DWORD dwAddress,
   LPTSTR lpszHostName,
   INT nMaxLength
);
```

The **IcmpResolveAddress** function resolves a numeric IP address into a fully qualified domain name.

Parameters

dwAddress

The IP address to be resolved, specified as an unsigned 32-bit integer in network byte order.

lpszHostName

A buffer that will contain a null-terminated string that specifies the fully qualified domain name for the host. It is recommended that this buffer be at least 64 characters in length.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the host name. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpResolveAddress** function is used to perform a reverse DNS lookup, converting a numeric IP address into a fully qualified domain name. This is useful for functions like **IcmpTraceRoute**, which return host addresses as numeric values in network byte order. If the reverse DNS lookup fails because there is no PTR record for the given IP address, a printable form of the address in dotted notation will be returned in the string buffer.

Calling this function will cause the thread to block until the IP address is resolved, or until the DNS query times out because there is no reverse record for the address. In some cases, a reverse lookup can take several seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpGetHostAddress, IcmpGetHostName, IcmpSetHostAddress, IcmpSetHostName, IcmpTraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSendEcho Function

```
INT WINAPI IcmpSendEcho(
   HCLIENT hClient
);
```

The **IcmpSendEcho** function sends an ICMP echo datagram to the remote host.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

For asynchronous client sessions, this function returns immediately. Otherwise, the client enters a blocking state and waits for a reply from the remote host. The function returns when a reply has been received, or the operation times-out.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpEcho, IcmpGetSendCount, IcmpRecvEcho, IcmpSetSequenceId, IcmpSetTimeToLive, IcmpTraceRoute

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetHostAddress Function

```
INT WINAPI IcmpSetHostAddress(
    HCLIENT hClient,
    DWORD dwAddress
);
```

The **IcmpSetHostAddress** function specifies the IP address of the host to receive an ICMP echo datagram. If the address specifies a new host, the current client statistics are reset.

Parameters

hClient

A handle to the client session.

dwAddress

The IP address of the remote host as a 32-bit integer value, specified in network byte order.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

To specify a remote host name or an IP address as a string in dot notation, use the **IcmpSetHostName** function instead.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetHostAddress, IcmpGetHostName, IcmpSetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetHostName Function

```
INT WINAPI IcmpSetHostName(
   HCLIENT hClient,
   LPCTSTR lpszHostName
);
```

The **IcmpSetHostName** function specifies the name of the host to receive an ICMP echo datagram. If the name specifies a new host, the current client statistics are reset.

Parameters

hClient

A handle to the client session.

lpszHostName

A null-terminated string which specifies the name, or IP address in dot-notation, of the remote host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpGetHostAddress, IcmpGetHostName, IcmpSetHostAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetLastError Function

```
VOID WINAPI IcmpSetLastError(
   DWORD dwErrorCode
);
```

The **IcmpSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those functions which call **IcmpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **IcmpGetLastError** function. The use of **IcmpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetErrorString, IcmpGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetPacketSize Function

```
UINT WINAPI IcmpSetPacketSize(
    HCLIENT hClient,
    UINT nPacketSize
);
```

The **IcmpSetPacketSize** function sets the size of the ICMP datagram packet that is sent to the remote host.

Parameters

hClient

Handle to the client session.

nPacketSize

Size of the ICMP datagram packet in bytes. The minimum packet size is 1 byte and the maximum packet size is 65,535 bytes.

Return Value

If the function succeeds, the return value is the previous ICMP datagram packet size. If the function fails, the return value is zero. To get extended error information, call **IcmpGetLastError**.

Remarks

Note that packet sizes over 512 bytes may not be supported by your networking software or configuration. It is recommended that most applications use the minimum packet size of 32 bytes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpSetSequenceId, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetSequenceId Function

```
INT WINAPI IcmpSetSequenceId(
    HCLIENT hClient,
    INT nSequenceId
);
```

The **IcmpSetSequenceId** function sets the sequence identifier for the next ICMP echo datagram sent by the client. The default sequence identifier for the first datagram is one.

Parameters

hClient

A handle to the client session.

nSequenceId

The sequence identifier for the next datagram sent by the client.

Return Value

If the function succeeds, the return value is the previous sequence identifier. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetRecvCount, IcmpGetSendCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpGetTripTime, IcmpSetPacketSize, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetTimeout Function

```
DWORD WINAPI IcmpSetTimeout(
    HCLIENT hClient,
    DWORD dwTimeout
);
```

The **IcmpSetTimeout** function sets the number of milliseconds the client will wait for a response from the remote host. Once the specified number of milliseconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

dwTimeout

The number of milliseconds until a blocking operation fails.

Return Value

If the function succeeds, the return value is the value of the timeout before the function was called. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpCreateHandle, IcmpGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpSetTimeToLive Function

```
INT WINAPI IcmpSetTimeToLive(
    HCLIENT hClient,
    INT nTimeToLive
);
```

The **IcmpSetTimeToLive** function sets the maximum time-to-live for the next ICMP datagram sent by the client.

Parameters

hClient

A handle to the client session.

nTimeToLive

The time-to-live value for the next ICMP echo datagram.

Return Value

If the function succeeds, the return value is the previous time-to-live value. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded.

Calling this function changes the default TTL value for all subsequent ICMP datagrams sent by the library, with the default value being 255. Note that not all Windows Sockets implementations support setting the time-to-live value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpGetPacketSize, IcmpGetRecvCount, IcmpGetSendCount, IcmpGetSequenceId, IcmpGetTimeToLive, IcmpGetTripTime, IcmpSetPacketSize, IcmpSetSequenceId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpTraceRoute Function

```
INT WINAPI IcmpTraceRoute(
   LPCTSTR lpszHostName,
   UINT nMaxHops,
   DWORD dwTimeout,
   DWORD dwReserved,
   LPICMPTRACE lpTrace,
   LPDWORD lpdwSize
);
```

The **IcmpTraceRoute** function sends a series of ICMP echo datagrams to trace the route taken from the local system to the remote host.

Parameters

lpszHostName

A null-terminated string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nMaxHops

An unsigned integer which specifies the maximum number of routers the datagram will be forwarded through (the number of hops) to the remote host. The minimum value is 1 and the maximum value is 255. It is recommended that most applications specify a value of at least 30.

dwTimeout

An unsigned integer which specifies the number of milliseconds the function will wait for a response to an echo datagram.

dwReserved

A reserved parameter. This value should always be zero.

lpTrace

A pointer to an array of ICMPTRACE structures which will contain information about each intermediate host between the local and remote system. The number of structures must be at least the same as the maximum number of hops specified by the ***nMaxHops*** parameter. To determine the total number of bytes that must be allocated for the structure array, this parameter can be passed as NULL and the total size will be returned in the ***lpdwSize*** parameter.

lpdwSize

A pointer to an unsigned integer which should be initialized to the size of the ICMPTRACE array passed to the function. If the ***lpTrace*** parameter is NULL, the function will calculate the size of the array buffer that must be allocated and return that value. If both the ***lpTrace*** and ***lpdwSize*** parameters are NULL, no statistical information about the intermediate hosts will be collected.

Return Value

If the function succeeds, the return value is the number of intermediate hosts between the local and remote system. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpTraceRoute** function sends a series of ICMP echo datagrams to the specified host, adjusting the time-to-live value to determine the intermediate hosts that route the packet. The function returns the number of hops to the remote host.

It is important to note that the failure of an intermediate host to respond to an ICMP echo datagram may not indicate a problem with the remote system. Systems can be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

IcmpCreateHandle, IcmpEcho, IcmpRecvEcho, IcmpResolveAddress, IcmpSendEcho, IcmpSetTimeToLive

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IcmpUninitialize Function

```
VOID WINAPI IcmpUninitialize();
```

The **IcmpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **IcmpInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **IcmpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **IcmpUninitialize** for every successful call to **IcmpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csicmv12.lib

See Also

IcmpCloseHandle, IcmpCreateHandle, IcmpInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Control Message Protocol Data Structures

- ICMPTIME
- ICMPTRACE
- INTERNET_ADDRESS
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ICMPTIME

This structure is used by the IcmpGetTripTime function to return the round-trip times of an ICMP datagram.

```
typedef struct _ICMPTIME {
    DWORD dwTripAverage;
    DWORD dwTripMaximum;
    DWORD dwTripMinimum;
    DWORD dwTripTime;
} ICMPTIME, *LPICMPTIME;
```

Members

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the current host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to current host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the current host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the current host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ICMPTRACE

This structure is used by the IcmpTraceRoute function to return the route of an ICMP datagram.

```
typedef struct _ICMPTRACE
{
    UINT        nDistance;
    DWORD       dwHostAddress;
    DWORD       dwTripAverage;
    DWORD       dwTripMaximum;
    DWORD       dwTripMinimum;
} ICMPTRACE, *LPICMPTRACE;
```

Members

nDistance

The distance from the local host to the remote host for this route.

dwHostAddress

An unsigned integer which specifies the IP address of the remote host in network byte order.

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to specified host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the specified host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
     INT    ipFamily;
     BYTE   ipNumber[16];
}  INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol Library

Manage email messages and mailboxes on a mail server.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSMAPV12.DLL
Version	12.0.3024.1286
LibID	8A55FC0D-5E12-4932-9E86-1F88C1DF5024
Import Library	CSMAPV12.LIB
Dependencies	None
Standards	RFC 3501

Overview

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP.

The SocketTools IMAP library implements the current standard for this protocol, and provides functions to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The API is designed as a superset of the Post Office Protocol API, so developers who are used to working with the POP3 library will find the IMAP library very easy to integrate into an existing application.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy

development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol Functions

Function	Description
ImapAsyncConnect	Connect asynchronously to the specified server
ImapAttachThread	Attach the specified client handle to another thread
ImapCancel	Cancel the current blocking operation
ImapCheckMailbox	Check the current mailbox for new messages
ImapCloseMessage	Close the current message
ImapCommand	Send a command to the server
ImapConnect	Connect to the specified server
ImapCopyMessage	Copy a message from the current mailbox to another mailbox
ImapCreateMailbox	Create a new mailbox on the server
ImapCreateMessage	Create a new message in the specified mailbox
ImapCreateSecurityCredentials	Allocate a structure to establish client security credentials
ImapDecodeMailboxName	Decode a UTF-7 encoded mailbox name using the specified code page
ImapDeleteMailbox	Delete the specified mailbox from the server
ImapDeleteMessage	Delete the specified message from the mailbox
ImapDeleteSecurityCredentials	Delete the specified client security credentials
ImapDisableEvents	Disable the event notification mechanism
ImapDisableTrace	Disable logging of network function calls
ImapDisconnect	Disconnect from the current server
ImapEnableEvents	Enable the client event notification mechanism
ImapEnableTrace	Enable logging of network function calls to a file
ImapEncodeMailboxName	Encode a mailbox name using modified UTF-7 encoding
ImapEnumMessages	Enumerate the messages in the current mailbox
ImapEventProc	Callback function that processes events generated by the client
ImapExamineMailbox	Select the specified mailbox in read-only mode
ImapExpungeMailbox	Remove messages that have been marked for deletion from the current mailbox
ImapFreezeEvents	Suspend and resume event handling by the client
ImapGetCapability	Return a string which identifies the capabilities of the server
ImapGetCurrentMailbox	Return the name of the currently selected mailbox
ImapGetDeletedMessages	Return the messages that have been marked for deletion
ImapGetErrorString	Return a description for the specified error code
ImapGetFirstMailbox	Return the first mailbox according to specified criteria
ImapGetHeaderValue	Return the value of the specified header field
ImapGetHeaderValueEx	Return the value of a header field in the specified message part

ImapGetIdleThreadId	Return the ID of the thread created to monitor the client session
ImapGetLastError	Return the last error code
ImapGetMailboxDelimiter	Get the path delimiter for the specified mailbox hierarchy
ImapGetMailboxSize	Return the size of the specified mailbox
ImapGetMailboxStatus	Return the status of the specified mailbox
ImapGetMailboxUid	Return the unique identifier for the specified mailbox
ImapGetMessage	Retrieve the specified message from the server
ImapGetMessageCount	Return the number of messages available in the mailbox
ImapGetMessageCountEx	Return the number of messages available in the mailbox
ImapGetMessageFlags	Return the status flags for the specified message
ImapGetMessageHeaders	Retrieve the specified message header from the server
ImapGetMessageHeadersEx	Retrieve the specified message header, UID and flags from the server
ImapGetMessageId	Return the message ID string for the specified message
ImapGetMessageParts	Return the number of MIME message parts in the specified message
ImapGetMessageSender	Return the address of the message sender
ImapGetMessageSize	Return the size of the specified message
ImapGetMessageUid	Return the unique identifier for the specified message
ImapGetNewMessages	Return a list of the new messages in the current mailbox
ImapGetNextMailbox	Return the next mailbox name on the server
ImapGetResultCode	Return the result code from the previous command
ImapGetResultString	Return the result string from the previous command
ImapGetSecurityInformation	Return security information about the current client connection
ImapGetStatus	Return the current status of the client
ImapGetTimeout	Return the number of seconds until an operation times out
ImapGetTransferStatus	Return data transfer statistics
ImapGetUnseenMessages	Return a list of messages from the current mailbox that have not been read
ImapIdle	Enables mailbox status monitoring for the client session
ImapIdleProc	Callback function that receives update notifications from the server
ImapInitialize	Initialize the library and validates the specified license key at runtime
ImapIsBlocking	Determine if the client is blocked, waiting for information
ImapIsConnected	Determine if the client is connected to the server
ImapIsReadable	Determine if data can be read from the server
ImapIsWritable	Determine if data can be written to the server
ImapLogin	Login to the server
ImapOpenMessage	Open the specified message for reading on the server
ImapOpenMessageEx	Open the specified message for reading on the server, with additional options

ImapRead	Read data returned by the server
ImapRegisterEvent	Register an event handler for the specified event
ImapRenameMailbox	Rename the specified mailbox
ImapReselectMailbox	Reselect the current mailbox and return updated status information
ImapSearchMailbox	Search the mailbox according to specified criteria
ImapSelectMailbox	Select the specified mailbox in read-write mode
ImapSetLastError	Set the last error code for the current thread
ImapSetMessageFlags	Set one or more status flags for the specified message
ImapSetTimeout	Set the number of seconds until an operation times out
ImapStoreMessage	Store the contents of a message to the specified file
ImapSubscribeMailbox	Subscribe to the specified mailbox
ImapUndeleteMessage	Undelete the specified message from the current mailbox
ImapUninitialize	Terminate use of the library by the application
ImapUnselectMailbox	Unselect the current mailbox and expunge any deleted messages
ImapUnsubscribeMailbox	Unsubscribe from the specified mailbox
ImapWrite	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapAsyncConnect Function

```
HCLIENT WINAPI ImapAsyncConnect(
   LPCTSTR lpszHostName,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **ImapAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **ImapConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the IMAP server.

nRemotePort

The port number the client should use to establish the connection. A value of zero specifies that default port 143 should be used, which is the standard port number assigned to the IMAP service. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
IMAP_OPTION_IDENTIFY	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
IMAP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior

	of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
IMAP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
IMAP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option is the same as specifying IMAP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
IMAP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the STARTTLS command.
IMAP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
IMAP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
IMAP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which is used to establish the client credentials for a secure connection to the server. The function **ImapCreateSecurityCredentials** can be used to

create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

hEventWnd

The handle to the asynchronous notification window. This window receives messages which notify the client of various asynchronous client events that occur. Specifying this parameter and a message identifier causes the connection to be non-blocking. If this parameter is NULL, then a blocking connection is established.

uEventMsg

The message identifier that is used when an asynchronous client event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **ImapGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the lParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking

	operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **ImapDisableEvents** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **ImapAttachThread** function.

Specifying the IMAP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapConnect, ImapDisableEvents, ImapDisconnect, ImapEnableEvents, ImapInitialize, ImapUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapAttachThread Function

```
DWORD WINAPI ImapAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **ImapAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **ImapAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **ImapAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **ImapCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **ImapUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapConnect, ImapDisconnect, ImapUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCancel Function

```
INT WINAPI ImapCancel(
   HCLIENT hClient
);
```

The **ImapCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When the **ImapCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCheckMailbox Function

```
INT WINAPI ImapCheckMailbox(
   HCLIENT hClient,
   UINT *lpnMessages,
   UINT *lpnUnseen
);
```

The **ImapCheckMailbox** function requests that the server create a checkpoint of the currently selected mailbox, and returns the current number of messages and unseen messages.

Parameters

hClient

Handle to the client session.

lpnMessages

A pointer to an unsigned integer value which will contain the number of messages in the currently selected mailbox when the function returns.

lpnUnseen

A pointer to an unsigned integer value which will contain the number of unseen messages in the currently selected mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When the client requests a checkpoint, the server may perform implementation-dependent housekeeping for that mailbox, such updating the mailbox on disk with the current state of the mailbox in memory. On some systems this command has no effect other than to update the client with the current number of messages in the mailbox.

This function actually sends two IMAP commands. The first is the CHECK command, followed by the NOOP command to poll for any new messages that have arrived. In addition to polling the server for new messages, this command can also be used to ensure the idle timer on the server does not expire and force a disconnect from the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCreateMailbox, ImapGetFirstMailbox, ImapGetMailboxStatus, ImapGetNextMailbox, ImapRenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCloseMessage Function

```
INT WINAPI ImapCloseMessage(
    HCLIENT hClient
);
```

The **ImapCloseMessage** function closes the current message.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapCloseMessage** function closes the current message. If there is any remaining data left to be read from the message, it will be read and discarded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCreateMessage, ImapOpenMessage, ImapOpenMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCommand Function

```
INT WINAPI ImapCommand(
   HCLIENT hClient,
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **ImapCommand** function sends a command to the server. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the function succeeds, it returns an IMAP result code. If the command was successful, it returns IMAP_RESULT_OK. A return value of IMAP_RESULT_CONTINUE indicates the command was accepted and the caller should proceed with the next command. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. The current version of the protocol which is supported by this library is version 4rev1 as defined in RFC 3501.

Use the **ImapGetResultCode** function to determine the result of the command that was sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetResultCode, ImapGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapConnect Function

```
HCLIENT WINAPI ImapConnect(
   LPCTSTR lpszHostName,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **ImapConnect** function is used to establish a connection with the server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the IMAP server.

nRemotePort

The port number the client should use to establish the connection. A value of zero specifies that default port 143 should be used, which is the standard port number assigned to the IMAP service. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
IMAP_OPTION_IDENTIFY	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
IMAP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
IMAP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
IMAP_OPTION_SECURE	This option specifies that a secure connection should

	be established with the server and requires that the server support the TLS protocol. This option is the same as specifying IMAP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
IMAP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
IMAP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
IMAP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
IMAP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
IMAP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which is used to establish the client credentials for a secure connection to the server. The function **ImapCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **ImapGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should

create a background worker thread and establish a connection by calling **ImapConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **ImapAttachThread** function.

Specifying the IMAP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDisconnect, ImapInitialize, ImapLogin

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCopyMessage Function

```
INT WINAPI ImapCopyMessage(
   HCLIENT hClient,
   UINT nMessageId,
   LPCTSTR lpszMailbox
);
```

The **ImapCopyMessage** function copies a message from the current mailbox to the specified mailbox. The message is appended to the mailbox, and the message flags and internal date are preserved.

Parameters

hClient

Handle to the client session.

nMessageId

The message identifier which specifies which message is to be copied to the mailbox. This value must be greater than zero and specify a valid message number.

lpszMailbox

A null-terminated string which specifies the name of the mailbox that the message will be copied to. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

Return Value

If the function succeeds, it returns a value of zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the mailbox does not exist, the function will fail. To create a new mailbox, use the **ImapCreateMailbox** function. A message can be copied within the same mailbox, in which case the server may flag it as a new message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateMailbox, ImapGetResultCode, ImapGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCreateMailbox Function

```
INT WINAPI ImapCreateMailbox(
   HCLIENT hClient,
   LPCTSTR lpszMailbox
);
```

The **ImapCreateMailbox** function creates a new mailbox on the server.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the new mailbox to be created.

Return Value

If the function succeeds, it returns a value of zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the mailbox name is suffixed with the server's hierarchy delimiter, this indicates to the server that the client intends to create mailbox names under the specified name in the hierarchy. If superior hierarchical names are specified in the mailbox name, then the server may automatically create them as needed. For example, if the mailbox name "Mail/Office/Projects" is specified and "Mail/Office" does not exist, it may be automatically created by the server.

The special mailbox name INBOX is reserved, and cannot be created. It is recommended that mailbox names only consist of printable ASCII characters, and the special characters "*" and "%" should be avoided.

If you need to create a mailbox which contains Unicode characters or symbols, it is recommended you always use the Unicode version of this function. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. The ANSI version of this function will only accept mailbox names using either modified UTF-7 encoding or UTF-8 encoding.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDecodeMailboxName, ImapDeleteMailbox, ImapEncodeMailboxName, ImapGetFirstMailbox, ImapGetNextMailbox, ImapRenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCreateMessage Function

```
INT WINAPI ImapCreateMessage(
   HCLIENT hClient,
   LPCTSTR lpszMailbox,
   DWORD dwReserved,
   LPVOID lpMessage,
   DWORD dwMessageSize,
   DWORD dwFlags
);
```

The **ImapCreateMessage** function creates a message, appending it to the contents of the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the name of the mailbox that the message will be created in. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

dwReserved

A reserved parameter. This value should always be zero.

lpMessage

The buffer which contains the message to be created.

dwMessageSize

An unsigned integer value which specifies the size of the message in bytes. If this value is zero, then it is assumed that the ***lpMessage*** parameter points to a string.

dwFlags

An unsigned integer that specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message is not completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message is flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the mailbox does not exist, the function will fail. To create a new mailbox, use the **ImapCreateMailbox** function. If the message is created in the mailbox that is currently selected, the server may flag it as recent.

This function is typically used by applications to store messages which have already been sent to a user. After a message has been delivered using the SMTP protocol, that same message may be created in a mailbox on the IMAP server so that the user has access to those messages.

If the Unicode version of this function is called and the ***dwMessageSize*** parameter is zero, the ***lpMessage*** parameter must point to a wide string terminated with a null character. The string will be automatically converted to UTF-8 when the message is created. In all other cases, the ***lpMessage*** parameter is treated as a pointer to a byte array which contains the message contents and the ***dwMessageSize*** parameter specifies the number of bytes in the array.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCopyMessage, ImapDeleteMessage, ImapGetMessageFlags, ImapSetMessageFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapCreateSecurityCredentials Function

```
BOOL WINAPI ImapCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **ImapCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **ImapDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **ImapConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
ImapCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                              0,
                              NULL,
                              NULL,
                              lpszCertStore,
                              lpszCertName,
                              NULL,
                              &lpSecCred);

hClient = ImapConnect(lpszHostName,
                      IMAP_PORT_SECURE,
                      IMAP_TIMEOUT,
                      IMAP_OPTION_SECURE,
                      lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapConnect, ImapDeleteSecurityCredentials, ImapGetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDecodeMailboxName Function

```
INT WINAPI ImapDecodeMailboxName(
   INT nCodePage,
   LPCSTR lpszMailboxName,
   LPTSTR lpszDecodedName,
   INT nMaxLength
);
```

The **ImapDecodeMailboxName** function decodes a UTF-7 encoded mailbox name using the specified code page.

Parameters

nCodePage

An integer value which specifies the code page which should be used when decoding the mailbox name. If the ANSI version of this function is called, this parameter can be any valid code page, or the constant CP_ACP which specifies the current active code page for the locale. If the Unicode version of this function is called, this parameter should always be zero.

lpszMailboxName

A null-terminated string which contains the UTF-7 encoded name of the mailbox. The encoded string must use the modified UTF-7 format defined in the IMAP4 protocol standard. This parameter cannot be a NULL pointer or an empty string.

lpszDecodedName

A null-terminated string buffer which will contain the decoded mailbox name. This parameter cannot be NULL and must be large enough to contain the entire name.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the *lpszDecodedName* string buffer. This value must be greater than zero and the buffer must be large enough to store the entire name.

Return Value

If the function succeeds, it returns the number of characters in the decoded mailbox name. If an error occurs, the function returns zero. To get extended error information, call **ImapGetLastError**.

Remarks

Mailbox names which contain non-ASCII characters must be converted to a modified version of UTF-7 encoding as defined by the IMAP4 protocol. This encoding allows applications to reference mailbox names which contain Unicode characters and provides support for languages other than English. This function is used to convert a UTF-7 encoded mailbox name using the current locale, primarily for display purposes.

For example, if the *lpszMailboxName* parameter specifies the string **"&A5QDvw06A7kDvA0u-"**, this function will convert it to the Greek word **"Δοκιμή"** and return that value in the *lpszDecodedName* buffer. It is recommended you use the Unicode version of this function whenever possible to ensure the mailbox name is decoded correctly.

The ANSI version of this function should only be used with legacy applications which do not support Unicode. The decoded mailbox name will be converted using the code page specified by the *nCodePage* parameter. If the code page does not support the characters used in the mailbox name, this can result in an incorrect conversion. Using the previous example, the mailbox name will always

decode correctly if the Unicode version of the function is used, but the ANSI version of the function will only decode it correctly if the code page is specified as 1253 (Greek).

This function should not be used to convert strings between Unicode and UTF-7. The encoding format is specifically intended for use with the IMAP4 protocol and should never be used for general purpose encoding or decoding. If you need to convert a UTF-7 encoded string to Unicode, use the Win32 API function **MultiByteToWideChar**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateMailbox, ImapEncodeMailboxName, ImapGetFirstMailbox, ImapGetNextMailbox, ImapRenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDeleteMailbox Function

```
INT WINAPI ImapDeleteMailbox(
   HCLIENT hClient,
   LPCTSTR lpszMailbox
);
```

The **ImapDeleteMailbox** function deletes a mailbox on the server.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the mailbox to be deleted.

Return Value

If the function succeeds, it returns value of zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

A mailbox cannot be deleted if it contains inferior hierarchical names and has the IMAP_FLAG_NOSELECT attribute. On most systems this is the case when the mailbox name references a directory on the server, and that directory contains other subdirectories or mailboxes. To remove the mailbox, you must first delete any child mailboxes that exist.

If the mailbox that is deleted is the currently selected mailbox, it will be automatically unselected and any messages marked for deletion will be expunged before the mailbox is removed. If the delete operation fails, the client will remain in an unselected state until either the **ImapExamineMailbox** or **ImapSelectMailbox** function is called.

The special mailbox name INBOX is reserved, and cannot be deleted.

If you need to delete a mailbox which contains Unicode characters or symbols, it is recommended you always use the Unicode version of this function. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. The ANSI version of this function will accept mailbox names using either modified UTF-7 encoding or UTF-8 encoding.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateMailbox, ImapGetFirstMailbox, ImapGetNextMailbox, ImapRenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDeleteMessage Function

```
INT WINAPI ImapDeleteMessage(
   HCLIENT hClient,
   UINT nMessageId
);
```

The **ImapDeleteMessage** function marks the specified message for deletion from the current mailbox.

Parameters

hClient

Handle to the client session.

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

This function only marks the message for deletion. The message is not actually deleted until the mailbox is expunged or another mailbox is selected. This function will return an error if the current mailbox is in read-only mode, such as if it was selected using the **ImapExamineMailbox** function.

It is important to note that unlike the POP3 protocol, a message that is marked for deletion is still accessible on the IMAP server until the mailbox is expunged. This means, for example, that a deleted message can still be retrieved using the **ImapGetMessage** function.

To determine if a message has been marked for deletion, use the **ImapGetMessageFlags** function and check if the IMAP_FLAG_DELETED bit flag has been set. To list all of the deleted messages in the current mailbox, use the **ImapGetDeletedMessages** function.

To remove the deletion flag from the message, use the **ImapUndeleteMessage** function. To prevent all messages in the current mailbox from being expunged, use the **ImapReselectMailbox** function to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetDeletedMessages, ImapGetMessage, ImapGetMessageCount, ImapGetMessageFlags, ImapReselectMailbox, ImapUndeleteMessage, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDeleteSecurityCredentials Function

```
VOID WINAPI ImapDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **ImapDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)
    ImapDeleteSecurityCredentials(&lpSecCred);

ImapUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateSecurityCredentials, ImapUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDisableEvents Function

```
INT WINAPI ImapDisableEvents(
    HCLIENT hClient
);
```

The **ImapDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapDisableEvents** function is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this function has been called.

This function is automatically called if the client has event notification enabled, and the **ImapDisconnect** function is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapEnableEvents, ImapRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDisableTrace Function

```
BOOL WINAPI ImapDisableTrace();
```

The **ImapDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapDisconnect Function

```
INT WINAPI ImapDisconnect(
   HCLIENT hClient
);
```

The **ImapDisconnect** function terminates the connection with the server, releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If a mailbox is selected at the time that **ImapDisconnect** is called, the current mailbox will be unselected and any messages that are marked for deletion will be expunged. To prevent any deleted messages from being removed from the mailbox, use the **ImapUnselectMailbox** function to unselect the current mailbox prior to disconnecting from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapConnect, ImapUninitialize, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapEnableEvents Function

```
INT WINAPI ImapEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **ImapEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **ImapRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. To avoid conflict with standard Windows messages, this value must be greater than WM_USER (1024) or an error will be returned. If the *hEventWnd* parameter is NULL, this value should be WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapEnableEvents** function is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer. The client must create a window message handler, which processes the various events. The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event identifier, and the high word will contain any error code. If no error has occurred, the high word will have a value of zero. The following events may be generated:

Value	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will be

	posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the lParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

It is not required that the client be placed in asynchronous (non-blocking) mode in order to receive event notifications, except for the connect, disconnect, read and write events. To disable event notification, call the **ImapDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDisableEvents, ImapRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapEnableTrace Function

```
BOOL WINAPI ImapEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **ImapEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapEncodeMailboxName Function

```
INT WINAPI ImapEncodeMailboxName(
   INT nCodePage,
   LPCTSTR lpszMailboxName,
   LPTSTR lpszEncodedName,
   INT nMaxLength
);
```

The **ImapEncodeMailboxName** function encodes a mailbox name using modified UTF-7 encoding.

Parameters

nCodePage

An integer value which specifies the code page which should be used when converting the mailbox name to Unicode. If the ANSI version of this function is called, this parameter can be any valid code page or a value of zero, which specifies the current active code page for the locale. If the Unicode version of this function is called, no conversion is required and this parameter should always be zero.

lpszMailboxName

A null-terminated string which contains the name of the mailbox. If the ANSI version of this function is called, this string is automatically converted to Unicode using the code page specified by the *nCodePage* parameter. This parameter cannot be a NULL pointer or an empty string.

lpszEncodedName

A null-terminated string buffer which will contain the UTF-7 encoded mailbox name. This parameter cannot be NULL and must be large enough to contain the entire name. The string will be null-terminated and will only contain printable ASCII characters.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the *lpszEncodedName* string buffer. This value must be greater than zero and the buffer must be large enough to store the entire name.

Return Value

If the function succeeds, it returns the number of characters in the encoded mailbox name. If an error occurs, the function returns zero. To get extended error information, call **ImapGetLastError**.

Remarks

Mailbox names which contain non-ASCII characters must be converted to a modified version of UTF-7 encoding as defined by the IMAP4 protocol. This encoding allows applications to reference mailbox names which contain Unicode characters and provides support for languages other than English. This function is used to convert a mailbox name which contains non-ASCII characters to the UTF-7 encoded format required by IMAP servers.

For example, if the *lpszMailboxName* parameter specifies the string **"Δοκιμή"**, this function will encode it as **"&A5QDvw06A7kDvA0u-"** and return that value in the *lpszEncodedName* buffer. It is recommended you use the Unicode version of this function whenever possible to ensure the mailbox name is encoded correctly.

The ANSI version of this function should only be used with legacy applications which do not support Unicode. The mailbox name will be converted to Unicode using the code page specified by the *nCodePage* parameter. If the code page does not support the characters used in the mailbox name,

this can result in an incorrect conversion. Using the previous example, the mailbox name will always encode correctly if the Unicode version of the function is used, but the ANSI version of the function will only encode it correctly if the code page is specified as 1253 (Greek).

If the ANSI version of this function is used and the ***nCodePage*** parameter is zero, this function will attempt to determine if the ***lpszMailboxName*** parameter specifies UTF-8 text as the mailbox name. If it does, the name will be converted from UTF-8 to modified UTF-7; otherwise, the name will be encoded as modified UTF-7 using the current active code page for the process.

This function should not be used to convert strings between Unicode and UTF-7. The encoding format is specifically intended for use with the IMAP4 protocol and should never be used for general purpose encoding or decoding. If you need to convert a Unicode string to UTF-7, use the Win32 API function **WideCharToMultiByte**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateMailbox, ImapDecodeMailboxName, ImapGetFirstMailbox, ImapGetNextMailbox, ImapRenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapEnumMessages Function

```
INT WINAPI ImapEnumMessages(
   HCLIENT hClient,
   UINT nFirstMessageId,
   UINT nLastMessageId,
   DWORD dwMessageFlags,
   DWORD dwReserved,
   LPIMAPMESSAGE lpMessageList,
   INT nMaxMessages
);
```

The **ImapEnumMessages** function enumerates the messages in the current mailbox, populating an array of IMAPMESSAGE structures which contain information about each message.

Parameters

hClient

Handle to the client session.

nFirstMessageId

An unsigned integer value which specifies the first message to enumerate. This value must be greater than zero and specify a valid message identifier. The first message in the mailbox has a value of one.

nLastMessageId

An unsigned integer value which specifies the last message to enumerate. This value must be greater than or equal to the value of the ***nFirstMessageId*** parameter. A special value of 0xFFFFFFFF can be used to specify the last message in the mailbox.

dwMessageFlags

This parameter is used to determine which messages are enumerated. If the value is zero, then all applicable messages will be enumerated. If the value is non-zero, only those messages which have at least one of the specified flags will be returned. More than one flag can be specified by using a bitwise operator.

Value	Description
IMAP_FLAG_ANSWERED	Return only those messages which have been answered.
IMAP_FLAG_DELETED	Return only those messages which have been marked for deletion.
IMAP_FLAG_DRAFT	Return only those messages which have been marked as draft copies.
IMAP_FLAG_URGENT	Return only those messages which have been flagged for urgent or special attention.
IMAP_FLAG_RECENT	Return only those messages which have been recently added to the mailbox.
IMAP_FLAG_SEEN	Return only those messages which have been read.

dwReserved

A reserved parameter. This value should always be set to zero.

lpMessageList

A pointer to an array of IMAPMESSAGE structures which will contain information about each of the messages returned by the server. This parameter cannot be NULL.

nMaxMessages

An integer value which specifies the maximum size of the IMAPMESSAGE array that was passed to the function. This value must be at least one.

Return Value

If the function succeeds, the return value is the number of messages that were enumerated. If no messages match the specified criteria, the function will return a value of zero. If an error is encountered, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the message UID is being stored locally by the client to identify the message over multiple sessions, it must also store the mailbox UID. Only the combination of the mailbox name, mailbox UID and message UID can be used to uniquely identify a given message on the server. Although IMAP server implementations are encouraged to maintain persistent message UIDs, they are not required to do so and those values may change if the mailbox UID changes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetMessage, ImapGetMessageCount, ImapGetMessageFlags, ImapGetMessageHeaders, ImapGetMessageSize, ImapGetMessageUid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapEnumMessageUids Function

```
INT WINAPI ImapEnumMessageUids(
   HCLIENT hClient,
   UINT nFirstMessageId,
   UINT nLastMessageId,
   DWORD dwMessageFlags,
   UINT* lpSequenceIdList,
   UINT* lpMessageUidList
   INT nMaxMessages
);
```

The **ImapEnumMessageUids** function populates an array of message sequence IDs and their corresponding message UIDs.

Parameters

hClient

Handle to the client session.

nFirstMessageId

An unsigned integer value which specifies the first message to enumerate. This value must be greater than zero and specify a valid message identifier. The first message in the mailbox has a value of one.

nLastMessageId

An unsigned integer value which specifies the last message to enumerate. This value must be greater than or equal to the value of the ***nFirstMessageId*** parameter. A special value of 0xFFFFFFFF (-1) can be used to specify the last message in the mailbox.

dwMessageFlags

This parameter is used to determine which messages are enumerated. If the value is zero, then all messages in the mailbox will be enumerated. If the value is non-zero, only those messages which have at least one of the specified flags will be returned. The following flags can be used to filter which messages are included in the result:

Value	Description
IMAP_FLAG_ANSWERED	Return only those messages which have been answered.
IMAP_FLAG_DELETED	Return only those messages which have been marked for deletion.
IMAP_FLAG_DRAFT	Return only those messages which have been marked as draft copies.
IMAP_FLAG_URGENT	Return only those messages which have been flagged for urgent or special attention.
IMAP_FLAG_RECENT	Return only those messages which have been recently added to the mailbox.
IMAP_FLAG_SEEN	Return only those messages which have been read.

lpSequenceIdList

A pointer to an array of unsigned integers that will receive the message sequence numbers for each enumerated message. The message sequence numbers are what most functions use to specify a

message in the current mailbox. These values should never be stored by your application. This parameter must not be NULL.

lpMessageUidList

A pointer to an array of unsigned integers that will receive the message UIDs corresponding to the messages in the sequence. The message UIDs are unique values which persist across multiple client sessions. These values can be stored by your application and then mapped to their corresponding sequential message ID. This parameter must not be NULL.

nMaxMessages

An integer value which specifies the maximum size of both arrays. This value cannot be zero.

Return Value

If the function succeeds, the return value is the number of entities populated in both the *lpSequenceIdList* and *lpMessageUidList* arrays. If no messages match the specified criteria, the function will return a value of zero. If an error is encountered, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapEnumMessageUids** function retrieves a list of messages in the specified range and populates parallel arrays with each message's sequence number and corresponding unique identifier (UID). This enables applications to correlate transient message numbers with persistent UIDs for more reliable tracking across sessions. The *lpSequenceIdList* and *lpMessageUidList* arrays must be the same size, and both must be large enough to store the number of values specified by the *nMaxMessages* parameter. Both the sequential and unique message IDs are unsigned integer values, but they cannot be used interchangeably. Note that functions such as **ImapGetMessage** and **ImapDeleteMessage** expect sequential message numbers, not message UIDs.

If your application stores the message UIDs to maintain some persistent information about the mailbox, this function will enable you to lookup those message UIDs and find the corresponding sequential message number. In most cases, you would want to call this function immediately after calling **ImapSelectMailbox**. This will cause the client to request a list of message UIDs in the selected mailbox within the range that you specify. Mailboxes with a very large number of messages may cause the application to block for an extended period of time while the list is returned and processed.

To map all messages in the selected mailbox, set `nFirstMessageId = 1`, `nLastMessageId = (UINT)-1` and `dwMessageFlags = 0`.

If the message UID is being stored locally by the client to identify the message over multiple sessions, it must also store the mailbox UID. Only the combination of the mailbox name, mailbox UID and message UID can be used to uniquely identify a given message on the server. Although IMAP server implementations are encouraged to maintain persistent message UIDs, they are not required to do so. Message UIDs may change if the mailbox is recreated or its unique identifier (UIDVALIDITY) is reset.

In some cases, particularly with newly created or infrequently used mailboxes, the message sequence numbers may *coincidentally* match the UIDs assigned by the server. For example, the first message in a new mailbox might have both a message sequence number and UID of 1. However, this is not guaranteed, and applications should never assume that sequence numbers and UIDs are interchangeable. Sequence numbers are dynamic and reflect the current ordering of messages in the selected mailbox. They can change between sessions or when messages are added or deleted. UIDs, on the other hand, are intended to be persistent identifiers that remain associated with a specific message, even if its sequence number changes.

It is important to note that a message UID is not the same as the Message-ID header field found

within the message content. The UID is a unique numeric identifier assigned by the IMAP server to a message in a specific mailbox. It is used by the protocol to track messages even as their sequence numbers change due to mailbox activity (e.g.: new messages arriving or messages being deleted). In contrast, the Message-ID header is a string value generated by the sending email client when the message is created. It is part of the message headers and intended to be globally unique for email threading and deduplication purposes, but it has no direct relationship to the UID used by the server.

Example

This example demonstrates how to enumerate all messages in the currently selected mailbox and retrieve the mapping between their sequential message numbers and server-assigned UIDs. This is useful when your application needs to track messages persistently across sessions using UIDs, while still interacting with the API functions that require sequence numbers such as **ImapGetMessage**.

```
// Get some information about the currently selected mailbox
UINT nLastMessage = 0;
DWORD dwMailboxSize = 0;
INT nMessageCount = ImapGetMessageCountEx(hClient, &nLastMessage, &dwMailboxSize);

_tprintf(_T("There are %d messages in %s (last message is %u, mailbox size is %lu
bytes)\n"),
         nMessageCount, szMailbox, nLastMessage, dwMailboxSize);

if (nMessageCount < 1)
{
    // There are no messages in the current mailbox
    return;
}

// Allocate memory for the array of sequence numbers and UIDs
UINT *pSeqList = (UINT *)malloc(nMessageCount * sizeof(UINT));
if (pSeqList == NULL)
{
    // Allocation failed
    return;
}

UINT *pUidList = (UINT *)malloc(nMessageCount * sizeof(UINT));
if (pUidList == NULL)
{
    free(pSeqList); // Clean up previously allocated memory
    return;
}

// Populate the arrays with message sequence numbers and corresponding UIDs
INT nUidCount = ImapEnumMessageUids(
                    hClient,        // Handle to the client session
                    1,              // First message
                    (UINT)-1,       // Last message (all messages)
                    0,              // Any message flags
                    pSeqList,       // Array of sequential message IDs
                    pUidList,       // Array of unique message IDs
                    nMessageCount); // Maximum number of IDs in the arrays

if (nUidCount > 0 && nUidCount <= nMessageCount)
{
    // Display the mapping of message sequence numbers to UIDs
    for (INT nIndex = 0; nIndex < nUidCount; nIndex++)
```

```
    {
        _tprintf(_T("\tMessage %u was assigned UID %u\n"), pSeqList[nIndex],
pUidList[nIndex]);
    }
}
else
{
    // Enumeration failed or no matching messages
    DWORD dwError = ImapGetLastError();
    _tprintf(_T("Unable to enumerate message UIDs (error code: 0x%lx)\n"),
dwError);
}

// Free the memory allocated for our arrays
free(pSeqList);
free(pUidList);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapEnumMessages, ImapGetMessage, ImapGetMessageCount, ImapGetMessageUid, ImapSelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapEventProc Function

```
VOID CALLBACK ImapEventProc(
   HCLIENT hClient,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **ImapEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **ImapRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **ImapRegisterEvent** function. The **ImapEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDisableEvents, ImapEnableEvents, ImapFreezeEvents, ImapRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapExamineMailbox Function

```
INT WINAPI ImapExamineMailbox(
   HCLIENT hClient,
   LPCTSTR lpszMailbox,
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ImapExamineMailbox** function selects the specified mailbox for read-only access.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the new mailbox to be examined.

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapExamineMailbox** function is used to select a mailbox in read-only mode. Messages can be read, but they cannot be modified or deleted from the mailbox and new messages will not lose their status as new messages if they are accessed.

If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **ImapUnselectMailbox** function prior to examining the new mailbox.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **ImapExamineMailbox** again with the same mailbox name.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-write mode, use the **ImapSelectMailbox** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDeleteMailbox, ImapGetFirstMailbox, ImapGetMailboxStatus, ImapGetNextMailbox,

ImapRenameMailbox, ImapReselectMailbox, ImapSelectMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapExpungeMailbox Function

```
INT WINAPI ImapExpungeMailbox(
   HCLIENT hClient,
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ImapExpungeMailbox** function removes all messages marked for deletion from the current mailbox.

Parameters

hClient

Handle to the client session.

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapExpungeMailbox** function causes all messages marked for deletion to be removed from the mailbox. Note that this can cause the mailbox's UID to change, and potentially invalidate the current message UIDs. It is recommended that applications use the information returned in the IMAPMAILBOX structure to update any internal state information stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCreateMailbox, ImapExamineMailbox, ImapGetFirstMailbox, ImapGetNextMailbox, ImapRenameMailbox, ImapSelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapFreezeEvents Function

```
INT WINAPI ImapFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **ImapFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four events of the same type occur, once event handling is resumed only one of those events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDisableEvents, ImapEnableEvents, ImapRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetCapability Function

```
INT WINAPI ImapGetCapability(
   HCLIENT hClient,
   LPTSTR lpszCapability,
   INT nMaxLength
);
```

The **ImapGetCapability** function returns a string which identifies the capabilities of the IMAP server.

Parameters

hClient

Handle to the client session.

lpszCapability

A null-terminated string buffer that will contain one or more tokens separated by spaces which identify the capabilities of the IMAP server.

nMaxLength

The maximum length of the capability string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the capability string, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

This function returns a string that contains one or more tokens separated by whitespace. Each token identifies a capability of the server. The following table lists some of the common capabilities:

Capability	Description
ACL	The RFC 2086 ACL extension
BINARY	The RFC 3516 binary content extension
CHILDREN	The RFC 3348 child mailbox extension
ID	The RFC 2971 ID extension
IDLE	The RFC 2177 IDLE extension
LOGINDISABLED	The RFC 2595 TLS/SSL extension
LOGINREFERRALS	The RFC 2221 login referrals extension
MAILBOXREFERRALS	The RFC 2193 mailbox referrals extension
MULTIAPPEND	The RFC 3501 MULTIAPPEND extension
NAMESPACE	The RFC 2342 namespace Extension
QUOTA	The RFC 2087 QUOTA extension
STARTTLS	The RFC 2595 TLS/SSL extension
UNSELECT	The RFC 3691 UNSELECT extension

Additional capabilities may be supported by your server. Note that experimental or custom capabilities

are always prefixed with the letter X. A list of standard IMAP capabilities is maintained by the Internet Assigned Numbers Authority (IANA) at www.iana.org.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapConnect, ImapDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetCurrentMailbox Function

```
INT WINAPI ImapGetCurrentMailbox(
    HCLIENT hClient,
    LPTSTR lpszMailbox,
    INT nMaxLength
);
```

The **ImapGetCurrentMailbox** function returns the name of the current mailbox that has been selected.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string buffer that will contain the current mailbox name. If the Unicode version of this function is called, the mailbox name will automatically be converted from modified UTF-7 encoding to Unicode. If the ANSI version of this function is called, the name will be returned using modified UTF-7 encoding. This parameter cannot be NULL and the buffer must be large enough to store the complete mailbox name.

nMaxLength

The maximum length of the mailbox string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the current mailbox name, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

It is recommended you use the Unicode version of this function, particularly if it is possible that mailbox names will contain Unicode characters or symbols. If the ANSI version of this function is called and a mailbox name contains Unicode characters, the name will be returned using a modified UTF-7 encoding format as defined by the IMAP4 standard. The mailbox name will not be returned as UTF-8 Unicode or as ANSI text converted to the current locale. If the Unicode version of this function is called, mailbox names will be automatically converted from modified UTF-7 to UTF-16 Unicode.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapExamineMailbox, ImapGetFirstMailbox, ImapGetNextMailbox, ImapSelectMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetDeletedMessages Function

```
INT WINAPI ImapGetDeletedMessages(
    HCLIENT hClient,
    UINT *lpnMessageIds,
    INT nMaxMessages
);
```

The **ImapGetDeletedMessages** function returns the message identifiers for those messages that have been marked for deletion in the current mailbox.

Parameters

hClient

Handle to the client session.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which have been marked for deletion in the current mailbox. This parameter may be NULL, in which case the function will return the number of deleted messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of messages marked for deletion in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected.

To remove the deleted flag from a message, use the **ImapUndeleteMessage** function. To prevent all messages in the current mailbox from being expunged, use the **ImapReselectMailbox** function to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetMessageCount, ImapGetMessageFlags, ImapGetNewMessages, ImapGetUnseenMessages, ImapSearchMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetErrorString Function

```
INT WINAPI ImapGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **ImapGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **ImapGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description should be returned. If this value is zero, then the description of the last error will be returned. If the last error code is zero, indicating no error, then this function will return zero.

lpszDescription

The buffer that will contain a description of the specified error code. It is recommended that this buffer be at least 128 characters in length. If a NULL pointer is specified, then no message will be returned but the function will return the length of the error string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetLastError, ImapGetResultCode, ImapGetResultString, ImapSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetFirstMailbox Function

```
INT WINAPI ImapGetFirstMailbox(
   HCLIENT hClient,
   LPCTSTR lpszReference,
   LPCTSTR lpszWildcard,
   DWORD dwOptions,
   LPVOID lpvReserved,
   LPTSTR lpszMailbox,
   INT nMaxLength
   LPDWORD lpdwFlags
);
```

The **ImapGetFirstMailbox** function returns the name of the first matching mailbox.

Parameters

hClient

Handle to the client session.

lpszReference

A null-terminated string which specifies the reference name. An empty string or NULL pointer specifies that the default mailbox hierarchy for the current user is returned. If the reference name is provided, this must be the name of a mailbox or a level of the mailbox hierarchy which provides the context in which the mailbox name is interpreted.

lpszWildcard

A null-terminated string which specifies the mailbox name to match. The wildcard character "*" may be used to match any portion of the mailbox hierarchy, including the delimiter. The wildcard character "%" matches any portion of the mailbox name, but does not match the mailbox delimiter. An empty string or NULL pointer specifies that all mailboxes in the context of the *lpszReference* parameter should be returned.

dwOptions

Specifies one or more options which controls how mailboxes are returned by the function. The options are bit flags which may be combined using a bitwise operator. One or more of the following values may be used:

Value	Description
IMAP_LIST_DEFAULT	This option specifies that all regular, selectable mailboxes should be returned.
IMAP_LIST_SUBSCRIBED	This option specifies that only subscribed mailboxes should be returned.
IMAP_LIST_FOLDERS	This option specifies that non-selectable mailbox folders should also be returned.
IMAP_LIST_HIDDEN	This option specifies that hidden mailboxes should be returned.
IMAP_LIST_INFERIOR	This option specifies that inferior mailboxes should be returned if an explicit wildcard mask is not specified.

lpvReserved

A reserved parameter. This value should always be set to NULL.

lpszMailbox

A null-terminated string buffer which will contain the first matching mailbox. This parameter cannot be NULL. A minimum buffer size of at least 128 character is recommended. If the Unicode version of this function is called, the mailbox name will automatically be converted from modified UTF-7 encoding to Unicode. If the ANSI version of this function is called, the mailbox name will be returned using modified UTF-7 encoding.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the first matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Value	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the function succeeds, it returns the length of the mailbox name. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetFirstMailbox** function is used to begin enumerating the available mailboxes for the current user on the IMAP server. Subsequent mailbox names are returned by calling **ImapGetNextMailbox** until the function returns IMAP_ERROR with an error code of ST_ERROR_NO_MORE_MAILBOXES.

It is recommended you use the Unicode version of this function, particularly if it is possible that mailbox names will contain Unicode characters or symbols. If the ANSI version of this function is called and a mailbox name contains Unicode characters, the name will be returned using a modified UTF-7 encoding format as defined by the IMAP4 standard. The mailbox name will not be returned as UTF-8 Unicode or as ANSI text converted to the current locale. If the Unicode version of this function is

called, mailbox names will be automatically converted from modified UTF-7 to UTF-16 Unicode.

The function of the *lpszReference* and *lpszWildcard* parameters are implementation dependent and generally are tied to the underlying operating system. On a UNIX based system, it can be helpful to think of the reference name as the directory where mailbox folders are stored, and the mailbox name as the name to search for in that directory and any subdirectories, if applicable. If the reference name is an empty string or NULL pointer, this typically refers to the current user's home directory.

Generally speaking, a reference name should only be specified if you or the user of the application knows the directory structure on the IMAP server. Incorrectly using a reference name can have serious negative side-effects. For example, specifying a reference name of "/" on a UNIX based system could cause the IMAP server to return search every directory on the system for a matching mailbox name. Similarly, the IMAP server may be unable to distinguish between regular files in the user's home directory and mailboxes. For this reason, most IMAP clients require that the user specify the directory on the server where their mailboxes are stored. Typically this is subdirectory named "mail" or "Mail" under the user's home directory. For non-UNIX servers, the mailbox hierarchy may represented differently, including a flat hierarchy.

Hidden mailboxes are those mailboxes which use the UNIX convention of the name beginning with a period. Therefore, a mailbox named ".secrets" would not normally be returned by the **ImapGetFirstMailbox** and **ImapGetNextMailbox** functions. The IMAP_LIST_HIDDEN option causes all mailboxes to be returned.

The IMAP_LIST_INFERIOR option will return inferior mailboxes (mailboxes located in folders or subdirectories) if a wildcard mask is not specified. If a wildcard mask is specified, this option has no effect and only those mailboxes which match the wildcard will be returned.

Subscribed mailboxes are those which were specified using the **ImapSubscribeMailbox** function. Marked mailboxes are typically those which have some special importance to the user.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDeleteMailbox, ImapGetCurrentMailbox, ImapGetMailboxStatus, ImapGetNextMailbox, ImapRenameMailbox, ImapSelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetHeaderValue Function

```
INT WINAPI ImapGetHeaderValue(
   HCLIENT hClient,
   UINT nMessageId,
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxLength
);
```

The **ImapGetHeaderValue** function returns the value of a header field in the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the header field value. If the header field is not present in the message, the function will return a value of zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetHeaderValue** function returns the value of header field from the specified message. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this function.

To return the value of a header field from a specific part of a multipart MIME message, use the **ImapGetHeaderValueEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetHeaderValueEx, ImapGetMessageHeaders, ImapGetMessageId, ImapGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetHeaderValueEx Function

```
INT WINAPI ImapGetHeaderValueEx(
    HCLIENT hClient,
    UINT nMessageId,
    UINT nMessagePart,
    LPVOID lpvReserved,
    LPCTSTR lpszHeader,
    LPTSTR lpszValue,
    INT nMaxLength
);
```

The **ImapGetHeaderValueEx** function returns the value of a header field from the specified message part.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that the header value will be be retrieved from. Message parts start with a value of one. A value of zero specifies that the RFC822 header field for the message will be used.

lpvReserved

A reserved parameter. This value should always be NULL.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the header field value. If the header field is not present in the message, the function will return a value of zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetHeaderValueEx** function returns the value of a header field from the specified message part. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this function.

To determine if a message is a multipart MIME message, use the **ImapGetMessageParts** function. The return value specifies the number of parts in the message, with a value greater than one

indicating that it is a multipart message.

Note that unlike the SocketTools MIME API which considers the first message part to be zero, the IMAP protocol defines the first message part to be one.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetHeaderValue, ImapGetMessageHeaders, ImapGetMessageId, ImapGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetIdleThreadId Function

```
DWORD WINAPI ImapGetIdleThreadId(
    HCLIENT hClient
);
```

The **ImapGetIdleThreadId** function returns ID of the thread that is monitoring the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, it returns the ID of the thread that is checking for update notifications from the server. If there is no active thread monitoring the client session, the function will return zero.

Remarks

The worker thread that monitors the client connection in the background can terminate if an IMAP command is sent to the server, if the **ImapCancel** function is called or if the client disconnects from the server. The **ImapGetIdleThreadId** function enables the application to determine if this background thread is still active or not.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapIdle, ImapIdleProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetLastError Function

```
DWORD WINAPI ImapGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last error code value. Functions set this value by calling the **ImapSetLastError** function. The return value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

You should call the **ImapGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **ImapSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those functions which call **ImapSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetErrorString, ImapSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMailboxDelimiter Function

```
INT WINAPI ImapGetMailboxDelimiter(
    HCLIENT hClient,
    LPCTSTR lpszMailbox,
    LPTSTR lpszDelimiter,
    INT nMaxLength
);
```

The **ImapGetMailboxDelimiter** function returns the hierarchical path delimiter used for the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the mailbox name. This parameter may be NULL or an empty string, in which case the default delimiter will be returned.

lpszDelimiter

A null-terminated string buffer that will contain the mailbox delimiter.

nMaxLength

The maximum length of the delimiter string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the delimiter for the specified mailbox, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the IMAP server supports multiple levels of mailboxes, then a special character or sequence of characters are used as delimiters between different levels of the mailbox hierarchy. On most systems, including most UNIX and Windows platforms, the delimiter is the forward slash "/" character.

It is possible that an IMAP server may only support a flat namespace, in which case this function will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapExamineMailbox, ImapSelectMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMailboxSize Function

```
DWORD WINAPI ImapGetMailboxSize(
    HCLIENT hClient,
    LPCTSTR lpszMailbox
);
```

The **ImapGetMailboxSize** function returns the size of the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the mailbox name.

Return Value

If the function succeeds, it returns the size of the mailbox. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMailboxSize** function may require a significant amount of time to calculate the mailbox size if there are a large number of messages in the mailbox. If the specified mailbox is not currently selected, then the current mailbox is unselected, the new mailbox is selected and the size calculated, and then the original mailbox is re-selected. This will have the side-effect of causing any messages marked for deletion to be expunged from the mailbox.

Because it can potentially result in long delays, it is not recommended that an application calculate the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateMailbox, ImapGetFirstMailbox, ImapGetNextMailbox, ImapRenameMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMailboxStatus Function

```
INT WINAPI ImapGetMailboxStatus(
   HCLIENT hClient,
   LPCTSTR lpszMailbox,
   LPIMAPMAILBOXSTATUS lpMailboxStatus
);
```

The **ImapGetMailboxStatus** function returns status information about the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string that specifies the mailbox to return information about.

lpMailboxStatus

A pointer to an IMAPMAILBOXSTATUS structure which contains status information about the specified mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMailboxStatus** function enables an application to obtain status information about a mailbox without having to select another mailbox or open a second connection to the IMAP server to examine the mailbox. The information returned is a subset of the information returned when a mailbox is selected.

If you need to check the status of a mailbox which contains Unicode characters or symbols, it is recommended you use the Unicode version of this function. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. The ANSI version of this function will only accept mailbox names using modified UTF-7 encoding or UTF-8 encoding.

Requesting information for a mailbox may be a slow operation, requiring the server to open the mailbox in read-only mode internally in order to obtain some of the status information. For this reason, this function should not be used to check for new messages; use the **ImapCheckMailbox** function instead.

Some IMAP servers may return an error if you attempt to obtain status information about the currently selected mailbox. The protocol standard states that clients should not use this method on the currently selected mailbox, and should instead use the information returned by the **ImapSelectMailbox** or **ImapExamineMailbox** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCheckMailbox, ImapCreateMailbox, ImapExamineMailbox, ImapSelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMailboxUid Function

```
DWORD WINAPI ImapGetMailboxUid(
   HCLIENT hClient,
   LPCTSTR lpszMailbox
);
```

The **ImapGetMailboxUid** function returns the unique identifier for the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string that specifies the mailbox name.

Return Value

If the function succeeds, it returns a non-zero value. If no unique identifier is assigned to the mailbox, the function will return zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMailboxUid** function returns an unsigned 32-bit value which uniquely identifies the mailbox and corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

An application can use the mailbox UID value in combination with the message UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetCurrentMailbox, ImapGetMailboxStatus, ImapGetMessageUid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessage Function

```
INT WINAPI ImapGetMessage(
   HCLIENT hClient,
   UINT nMessageId,
   UINT nMessagePart,
   LPVOID lpvMessage,
   LPDWORD lpdwLength,
   DWORD dwOptions
);
```

The **ImapGetMessage** function retrieves a message from the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

lpvMessage

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvMessage*** parameter. If the ***lpvMessage*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.
IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are

	used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessage** function is used to retrieve a message from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the *lpvMessage* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvMessage* parameter point to a global memory handle which will contain the message data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **ImapEnableEvents**, or by registering a callback function using the **ImapRegisterEvent** function.

To determine if a message is a multipart MIME message, use the **ImapGetMessageParts** function. The return value specifies the number of parts in the message, with a value greater than one indicating that it is a multipart message. Combining the IMAP_SECTION_HEADER and IMAP_SECTION_BODY options will only return the header and body for the specified message if the *nMessagePart* parameter is zero. Due to a limitation of the IMAP FETCH command, if a message part is specified then only the body of that message part will be returned.

Note that unlike the SocketTools MIME API which considers the first message part to be zero, the IMAP protocol defines the first message part to be one.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csmapv12.lib

See Also

ImapGetMessageCount, ImapGetMessageHeaders, ImapGetMessageParts, ImapStoreMessage

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageCount Function

```
INT WINAPI ImapGetMessageCount(
    HCLIENT hClient
);
```

The **ImapGetMessageCount** function returns the number of messages that are available in the currently selected mailbox.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, it returns the number of messages in the currently selected mailbox. If no messages are available, this function will return zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageCount** function does not distinguish between messages that are marked for deletion and those which are not. This is because messages that are marked for deletion on an IMAP server can still be accessed until the mailbox is expunged or unselected. This differs from the POP3 protocol, where messages cannot be accessed once they have been marked for deletion.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDeleteMessage, ImapGetHeaderValue, ImapGetMessage, ImapGetMessageCountEx, ImapGetMessageHeaders, ImapStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageCountEx Function

```
INT WINAPI ImapGetMessageCountEx(
   HCLIENT hClient,
   UINT *lpnLastMessage,
   DWORD *lpdwMailboxSize
);
```

The **ImapGetMessageCountEx** function returns the number of messages that are available in the currently selected mailbox, and optionally the size of the mailbox in bytes.

Parameters

hClient

Handle to the client session.

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. If a NULL value is specified, this argument is ignored.

Return Value

If the function succeeds, it returns the number of messages that are currently available. If no messages are available, this function will return zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageCountEx** function is provided for compatibility with the POP3 API. The *lpnLastMessage* parameter will always contain the same value returned by the function since there is no distinction between the message count and the last available message. This is because messages that are marked for deletion on an IMAP server can still be accessed until the mailbox is expunged or unselected. This differs from the POP3 protocol, where messages cannot be accessed once they have been marked for deletion.

If the *lpdwMailboxSize* parameter is specified, this function will call **ImapGetMailboxSize** to determine the size of the currently selected mailbox. Unlike the POP3 protocol, calculating the mailbox size may require a significant amount of time if there are a large number of messages in the mailbox. It is not recommended that an application request the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDeleteMessage, ImapGetHeaderValue, ImapGetMailboxSize, ImapGetMessage, ImapGetMessageHeaders, ImapStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageFlags Function

```
BOOL WINAPI ImapGetMessageFlags(
   HCLIENT hClient,
   UINT nMessageId,
   LPDWORD lpdwMessageFlags
);
```

The **ImapGetMessageFlags** function returns the message flags for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

lpdwMessageFlags

Pointer to an unsigned integer that will contain the message flags for the specified message. The value may be zero, or one or more of the following values:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the function succeeds, the return value is non-zero and the *lpdwMessageFlags* parameter contains the status flags for the specified message. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDeleteMessage, ImapGetMessageCount, ImapSetMessageFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageHeaders Function

```
INT WINAPI ImapGetMessageHeaders(
    HCLIENT hClient,
    UINT nMessageId,
    LPVOID lpvHeaders,
    LPDWORD lpdwLength
);
```

The **ImapGetMessageHeaders** function retrieves the headers for the specified message from the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

lpvHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvHeaders*** parameter. If the ***lpvHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessagHeaders** function is used to retrieve a message header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpvHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvHeaders*** parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **ImapEnableEvents**, or by registering a callback function using the **ImapRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetMessage, ImapGetMessageCount, ImapGetMessageParts, ImapStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageHeadersEx Function

```
INT WINAPI ImapGetMessageHeadersEx(
   HCLIENT hClient,
   UINT nMessageId,
   DWORD dwReserved,
   LPDWORD lpdwMessageUID,
   LPDWORD lpdwMessageFlags,
   LPVOID lpvHeaders,
   LPDWORD lpdwLength
);
```

The **ImapGetMessageHeadersEx** function retrieves the headers, unique ID and flags for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

dwReserved

A reserved parameter, this value should always be zero.

lpdwMessageUID

A pointer to an unsigned integer which will contain the unique ID assigned to the message by the server when the function returns. This parameter may be NULL if this information is not required.

lpdwMessageFlags

A pointer to an unsigned integer which will contain the message flags when the function returns. This parameter may be NULL if this information is not required, otherwise the value may be zero or one or more of the following bit flags:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

lpvHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvHeaders*** parameter. If the ***lpvHeaders*** parameter

points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessagHeadersEx** function is used to retrieve a message header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvHeaders* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

If the *lpdwMessageUID* and/or *lpdwMessageFlags* parameters are NULL, they will be ignored. Otherwise, the function will always initialize the values to zero and only sets them if the header block for the message can be retrieved.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **ImapEnableEvents**, or by registering a callback function using the **ImapRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetMessage, ImapGetMessageCount, ImapGetMessageFlags, ImapGetMessageParts, ImapStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageId Function

```
INT WINAPI ImapGetMessageId(
   HCLIENT hClient,
   UINT nMessageId,
   LPTSTR lpszMessageId,
   INT nMaxLength
);
```

The **ImapGetMessageId** function returns the message identifier string for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the length of the unique identifier string. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageId** function returns the message identifier from the Message-ID header of the specified message. The returned value is a string which can be used to identify a specific message, regardless if the message is moved to a different mailbox.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetHeaderValue, ImapGetMessage, ImapGetMessageHeaders, ImapGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageParts Function

```
INT WINAPI ImapGetMessageParts(
    HCLIENT hClient,
    UINT nMessageId
);
```

The **ImapGetMessageParts** function returns the number of parts in a MIME multipart message on the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the part count for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is the number of parts in the specified message. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageParts** function can be used to determine if a message on the server contains multiple parts. A multipart MIME message typically contains file attachments or multiple representations of the message, such as a version of the message in plain text and another using HTML markup.

If the function returns a value of one, then the message does not contain multiple parts and is a standard RFC822 formatted message. A value greater than one indicates that the message does have multiple parts. The **ImapGetMessageEx** function may be used to retrieve the data for a specific part of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetHeaderValue, ImapGetMessage, ImapGetMessageHeaders, ImapGetMessageId, ImapGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageSender Function

```
INT WINAPI ImapGetMessageSender(
   HCLIENT hClient,
   UINT nMessageId,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **ImapGetMessageSender** function returns the sender's address for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the address. If the sender cannot be determined, the function will return a value of zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageSender** function returns the email address specified in the Return-Path header field. This allows an application to be able to easily determine the sender, without parsing the header or downloading the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetHeaderValue, ImapGetMessageHeaders, ImapGetMessageId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageSize Function

```
DWORD WINAPI ImapGetMessageSize(
    HCLIENT hClient,
    UINT nMessageId
);
```

The **ImapGetMessageSize** function returns the size of the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is the size of the specified message in bytes. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetHeaderValue, ImapGetMessageHeaders, ImapGetMessageId, ImapGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetMessageUid Function

```
DWORD WINAPI ImapGetMessageUid(
    HCLIENT hClient,
    UINT nMessageId
);
```

The **ImapGetMessageUid** function returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to obtain the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the function will return zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageUid** function returns an unsigned integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque.

An application can use the message UID value in combination with the mailbox UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Although the IMAP4 protocol supports returning the UID assigned to a sequential message ID, there is no corresponding capability which does the reverse. If you wish to map a message UID back to the sequential message number in the mailbox, you can use the **ImapEnumMessageUids** function which will populate two arrays: one which contains the sequential message ID and a corresponding array which contains its UID.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapEnumMessageUids, ImapGetCurrentMailbox, ImapGetMailboxStatus, ImapGetMailboxUid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetNewMessages Function

```
INT WINAPI ImapGetNewMessages(
   HCLIENT hClient,
   UINT *lpnMessageIds,
   INT nMaxMessages
);
```

The **ImapGetNewMessages** function returns the message identifiers for those messages that have recently been added to the mailbox and have not been read.

Parameters

hClient

Handle to the client session.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have recently been added to the mailbox and have not been read. This parameter may be NULL, in which case the function will return the number of new messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of new messages in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **ImapGetMessage** or **ImapStoreMessage**, it is no longer considered to be a new message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetDeletedMessages, ImapGetMessageCount, ImapGetMessageFlags, ImapGetUnseenMessages, ImapSearchMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetNextMailbox Function

```
INT WINAPI ImapGetNextMailbox(
   HCLIENT hClient,
   LPTSTR lpszMailbox,
   INT nMaxLength
   LPDWORD lpdwFlags
);
```

The **ImapGetNextMailbox** function returns the name of the next matching mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string buffer which will contain the next matching mailbox. This parameter cannot be NULL. A minimum buffer size of at least 128 character is recommended. If the Unicode version of this function is called, the mailbox name will automatically be converted from modified UTF-7 encoding to Unicode. If the ANSI version of this function is called, the mailbox name will be returned using modified UTF-7 encoding.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the next matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Value	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the function succeeds, it returns the length of the mailbox name. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetNextMailbox** function returns the next matching mailbox name. When the last mailbox has been returned, the next call to this function will result in an error, with the last error code set to ST_ERROR_NO_MORE_MAILBOXES.

It is recommended you use the Unicode version of this function, particularly if it is possible that mailbox names will contain Unicode characters or symbols. If the ANSI version of this function is called and a mailbox name contains Unicode characters, the name will be returned using a modified UTF-7 encoding format as defined by the IMAP4 standard. The mailbox name will not be returned as UTF-8 Unicode or as ANSI text converted to the current locale. If the Unicode version of this function is called, mailbox names will be automatically converted from modified UTF-7 to UTF-16 Unicode.

Subscribed mailboxes are those which were specified using the **ImapSubscribeMailbox** function. Marked mailboxes are typically those which have some special importance to the user.

For more information about enumerating the available mailboxes on the IMAP server, refer to the **ImapGetFirstMailbox** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDeleteMailbox, ImapGetCurrentMailbox, ImapGetFirstMailbox, ImapGetMailboxStatus, ImapRenameMailbox, ImapSelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetResultCode Function

```
INT WINAPI ImapGetResultCode(
    HCLIENT hClient
);
```

The **ImapGetResultCode** function reads the result code returned by the server in response to a command. The result code is an integer value, and indicates if the operation succeeded or failed.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, it returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The following result codes may be returned by the IMAP server:

Value	Description
IMAP_RESULT_UNKNOWN	An unknown result code was returned by the server.
IMAP_RESULT_OK	The previous command completed successfully. The result string contains information about the results of the command.
IMAP_RESULT_NO	The previous command could not be completed. The result string contains information about why the command failed.
IMAP_RESULT_BAD	The previous command could not be completed, the command may be invalid or not supported on the server. The result string contains information about why the command failed.
IMAP_RESULT_CONTINUE	The command has executed and is waiting for additional data from the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCommand, ImapGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetResultString Function

```
INT WINAPI ImapGetResultString(
    HCLIENT hClient,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

The **ImapGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

The buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **ImapGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCommand, ImapGetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetSecurityInformation Function

```
BOOL WINAPI ImapGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **ImapGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (ImapGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapConnect, ImapCreateSecurityCredentials, ImapDeleteSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetStatus Function

```
INT WINAPI ImapGetStatus(
   HCLIENT hClient
);
```

The **ImapGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
IMAP_STATUS_IDLE	The client is current idle and not sending or receiving data.
IMAP_STATUS_CONNECT	The client is establishing a connection with the server.
IMAP_STATUS_READ	The client is reading data from the server.
IMAP_STATUS_WRITE	The client is writing data to the server.
IMAP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapIsBlocking, ImapIsConnected, ImapIsReadable, ImapIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetTimeout Function

```
INT WINAPI ImapGetTimeout(
   HCLIENT hClient
);
```

The **ImapGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapConnect, ImapIsReadable, ImapIsWritable, ImapRead, ImapSetTimeout, ImapWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetTransferStatus Function

```
INT WINAPI ImapGetTransferStatus(
   HCLIENT hClient,
   LPIMAPTRANSFERSTATUS lpStatus
);
```

The **ImapGetTransferStatus** function returns information about the current file transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an IMAPTRANSFERSTATUS structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetTransferStatus** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapEnableEvents, ImapGetStatus, ImapRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapGetUnseenMessages Function

```
INT WINAPI ImapGetUnseenMessages(
   HCLIENT hClient,
   UINT *lpnMessageIds,
   INT nMaxMessages
);
```

The **ImapGetUnseenMessages** function returns the message identifiers for those messages that have not been read.

Parameters

hClient

Handle to the client session.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have not been read. This parameter may be NULL, in which case the function will return the number of unseen messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of unseen messages in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **ImapGetMessage** or **ImapStoreMessage**, it is no longer considered to be an unseen message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetDeletedMessages, ImapGetMessageCount, ImapGetMessageFlags, ImapGetNewMessages, ImapSearchMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapIdle Function

```
INT WINAPI ImapIdle(
   HCLIENT hClient,
   UINT nTimeout,
   DWORD dwOptions,
   IMAPIDLEPROC lpfnIdleProc,
   DWORD_PTR dwParam
);
```

The **ImapIdle** function enables mailbox status monitoring for the client session, allowing the client to receive notifications from the server whenever a new message arrives or a message is expunged from the currently selected mailbox. This is typically used as an alternative to the client periodically polling the server for status information.

Parameters

hClient

Handle to the client session.

nTimeout

Specifies the timeout period in seconds to wait for a notification from the server. This parameter is only used when the IMAP_IDLE_WAIT option has been specified.

dwOptions

Specifies the options which should be used when enabling idle monitoring. The following options are supported:

Value	Description
IMAP_IDLE_NOWAIT	The function should return immediately after idle processing has been enabled. When this option is used, the application may continue to perform other functions while the client session is monitored for status updates sent by the server. The client will continue to monitor status changes until an IMAP command issued or the client disconnects from the server.
IMAP_IDLE_WAIT	The function should wait until the server sends a status update, or until the timeout period is reached. The client will stop monitoring status changes when the function returns. If this option is used in a single-threaded application, normal message processing can be impeded, causing the application to appear non-responsive until the timeout period is reached. It is strongly recommended that single-threaded applications with a user interface specify the IMAP_IDLE_NOWAIT option instead.

lpfnIdleProc

A pointer to an IMAPIDLEPROC callback function that will be invoked whenever the server sends an update notification to the client. This parameter must specify a valid function address and cannot be a NULL pointer.

dwParam

A user-defined value that is passed back to the caller whenever the callback function is invoked. This can be used to provide additional state information to the client. If it is not needed, the caller should

use a value of zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

Many IMAP servers support the ability to asynchronously send status updates to the client, rather than have the client periodically poll the server. The client enables this feature by calling **ImapIdle** and providing the address of a callback function that will be invoked whenever the server sends an update notification to the client. Typically these updates inform the client that a new message has arrived or that a message has been expunged from the mailbox.

The **ImapIdle** function can operate in two modes, based on the options specified by the caller. If the option IMAP_IDLE_NOWAIT is specified, the function begins monitoring the client session and returns control immediately to the caller. If the server sends a update notification, the callback function will be invoked with information about the status change. If the option IMAP_IDLE_WAIT is specified, the function will block waiting for the server to send a notification message to the client. The function will return when either a message is received or the timeout period is exceeded.

Sending an IMAP command to the server will cause the client to stop monitoring the session for status changes. To explicitly stop monitoring the session, use the **ImapCancel** function. To determine if the current client session is being monitored, use the **ImapGetIdleThreadId** function. A non-zero return value indicates that the client session is idle and being monitored.

This function works by sending the IDLE command to the server and starting a worker thread which monitors the connection and looks for untagged responses issued by the server. Callbacks will be invoked for EXISTS, EXPUNGE and RECENT messages. Note that some servers may periodically send untagged OK messages to the client, indicating that the connection is still active; these messages are explicitly ignored. Because the monitoring is performed in a different thread, the callback function is invoked in the context of that thread. Client event notifications are disabled while inside the callback function, and the **ImapIdle** function cannot be used to restart monitoring from within a callback function.

Applications should not perform any operation that takes a significant amount of time or updates the user interface from within the callback function. Instead, use flags or send application defined messages to indicate a change in state. For example, if the server sends a notification that a new email message has arrived, the application should not attempt to read the new message and update the user interface from within the callback function. Instead, it could use the **PostMessage** function to send an application-defined message to the UI thread indicating the change in state. The application would have a message handler for that Windows message and update the user interface, indicating that a new message has arrived.

An application should never make an assumption about how a particular server may send update notifications to the client. Servers can be configured to use different intervals at which notifications are sent. For example, a server may send new message notifications immediately, but may periodically notify the client when a message has been expunged. Alternatively, a server may only send notifications at fixed intervals, in which case the client would not be notified of any new messages until the interval period is reached. It is not possible for a client to know what a particular server's update interval is. Applications that require that degree of control should not use **ImapIdle** and should poll the server instead.

Example

```
// Begin monitoring the client session for status changes; when a new
// message arrives or a message is expunged, the UpdateHandler callback
// function will be invoked.
BOOL MonitorUpdates(HWND hwndNotify)
{
    INT nResult;

    nResult = ImapIdle(hClient, 0, IMAP_IDLE_NOWAIT, UpdateHandler, hwndNotify);

    if (nResult == IMAP_ERROR)
    {
        ShowError(ImapGetLastError());
        return FALSE;
    }

    return TRUE;
}

// This function is called whenever the server notifies the client
// that a new message has arrived or a message has been expunged from
// the current mailbox
#define WM_APP_NEWMESSAGE (WM_APP+1)
#define WM_APP_EXPUNGED   (WM_APP+2)

VOID CALLBACK UpdateHandler(HCLIENT hClient, UINT nUpdateId, UINT nMessageId,
DWORD_PTR dwParam)
{
    switch (nUpdateId)
    {
    case IMAP_UPDATE_MESSAGE:
        {
            // Send a message indicating that a new message has arrived
            PostMessage((HWND)dwParam, WM_APP_NEWMESSAGE, nMessageId, 0);
        }
        break;

    case IMAP_UPDATE_EXPUNGE:
        {
            // Send a message indicating that a message has been expunged
            PostMessage((HWND)dwParam, WM_APP_EXPUNGED, nMessageId, 0);
        }
        break;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetIdleThreadId, ImapIdleProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapIdleProc Function

```
VOID CALLBACK ImapIdleProc(
   HCLIENT hClient,
   UINT nUpdateId,
   UINT nMessageId,
   DWORD_PTR dwParam
);
```

The **ImapIdleProc** function is an application-defined callback function that is invoked whenever the server sends a status update to the client. For more information about status monitoring, refer to the **ImapIdle** function.

Parameters

hClient

Handle to the client session.

nUpdateId

An unsigned integer which specifies the type of update notification that has been sent by the server. It may be one of the following values:

Value	Description
IMAP_UPDATE_UNKNOWN	The server has sent an unrecognized notification message. The value of the ***nMessageId*** argument is undefined for this type of notification. This does not necessarily reflect an error condition, as some servers may send additional notification messages beyond the standard EXISTS, EXPUNGE and RECENT messages. Most applications should ignore this type of notification.
IMAP_UPDATE_MESSAGE	The server has sent notification message to the client indicating that a new message has arrived. The ***nMessageId*** argument will contain the message number for the new message. Typically this update notification occurs shortly after the new message has been stored in the current mailbox.
IMAP_UPDATE_EXPUNGE	The server has sent a notification message to the client indicating that a message has been removed from the current mailbox. The ***nMessageId*** argument will contain the message number for the message that has been removed. It is recommended that the application re-examine the mailbox when this notification is received. Typically this notification is only sent periodically by the server, and may not be sent immediately after a message has been expunged from the mailbox.
IMAP_UPDATE_MAILBOX	The server has sent notification message to the client indicating that the state of the mailbox has changed. The ***nMessageId*** argument is not used with this notification. This message is sent periodically by the server and may not be sent immediately after a new message arrives or a message is flagged as unread. It is recommended that the application re-

	examine the mailbox when this notification is received.

nMessageId

An unsigned integer which specifies the message number associated with the status change. Note that this argument is not used with the IMAP_UPDATE_MAILBOX notification and will contain a value of zero.

dwParam

A user-defined integer value which was specified when the **ImapIdle** function was called.

Return Value

None.

Remarks

An application must enable this callback function by passing its address to the **ImapIdle** function. The **ImapIdleProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetIdleThreadId, ImapIdle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapInitialize Function

```
BOOL WINAPI ImapInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **ImapInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **ImapInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **ImapUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **ImapInitialize** or **ImapUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapConnect, ImapDisconnect, ImapUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapIsBlocking Function

```
BOOL WINAPI ImapIsBlocking(
   HCLIENT hClient
);
```

The **ImapIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **ImapIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCancel, ImapIsConnected, ImapIsReadable, ImapIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapIsConnected Function

```
BOOL WINAPI ImapIsConnected(
    HCLIENT hClient
);
```

The **ImapIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapIsBlocking, ImapIsReadable, ImapIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapIsReadable Function

```
BOOL WINAPI ImapIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **ImapIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetStatus, ImapIsBlocking, ImapIsConnected, ImapIsWritable, ImapRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapIsWritable Function

```
BOOL WINAPI ImapIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **ImapIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetStatus, ImapIsBlocking, ImapIsConnected, ImapIsReadable, ImapWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapLogin Function

```
INT WINAPI ImapLogin(
   HCLIENT hClient,
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **ImapLogin** function authenticates the specified user in on the server. This function must be called after the connection has been established, and before attempting to retrieve messages or perform any other function on the server.

Parameters

hClient

Handle to the client session.

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Value	Description
IMAP_AUTH_LOGIN	Standard cleartext username and password is sent to the server. This authentication method is supported by all servers. Note that some servers may only support LOGIN authentication if a secure connection has been established.
IMAP_AUTH_PLAIN	Login using the PLAIN authentication mechanism as defined in RFC 4959. This authentication method is supported by most servers, although some may require that client establish a secure connection.
IMAP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the *lpszPassword* parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
IMAP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the *lpszPassword* parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
IMAP_AUTH_ANONYMOUS	Login using the anonymous Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4505. If this authentication method is specified, the *lpszUserName* parameter should specify a name or email address acceptable to the mail server. The *lpszPassword* parameter is

	ignored and may be NULL.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the current client session.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the current client session. If you are using the IMAP_AUTH_XOAUTH2 or IMAP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 bearer token provided by the mail service.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

In some cases, the user may be pre-authenticated by the server. In this case, the function will fail with the last error set to ST_ERROR_ALREADY_AUTHENTICATED. If a particular authentication method is not supported by the server, the last error will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE. For compatibility with the greatest number of servers, it is recommended that you use IMAP_AUTH_LOGIN as the authentication method.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The IMAP_AUTH_XOAUTH2 and IMAP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapConnect, ImapDisconnect, ImapInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapOpenMessage Function

```
INT WINAPI ImapOpenMessage(
   HCLIENT hClient,
   UINT nMessageId,
   DWORD dwReserved
);
```

The **ImapOpenMessage** function opens the specified message on the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapOpenMessage** function uses the FETCH command to access the specified message on the server. The client can then use the **ImapRead** function to read the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCloseMessage, ImapGetMessage, ImapGetMessageHeaders, ImapOpenMessageEx, ImapRead, ImapStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapOpenMessageEx Function

```
INT WINAPI ImapOpenMessageEx(
   HCLIENT hClient,
   UINT nMessageId,
   UINT nMessagePart,
   DWORD dwOffset,
   LPDWORD lpdwLength,
   DWORD dwOptions
);
```

The **ImapOpenMessageEx** function opens a message or a specific part of a multipart message in the current mailbox. The message data may also be limited a specific byte offset and length, which can be useful for previewing the contents.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

dwOffset

The byte offset into the message. This parameter can be used in conjunction with the ***lpdwLength*** parameter to return a specific part of a message. A value of zero specifies the beginning of the message.

lpdwLength

A pointer to an unsigned integer value which should be initialized to the maximum number of bytes to be read, and will contain the size of the message when the function returns. To specify the entire message, from the offset specified by the ***dwOffset*** parameter to the end of the message, initialize the ***lpdwLength*** parameter to a value of -1. This parameter may be NULL if the message size is not needed.

dwOptions

The low order word of this parameter specifies how the message data will be returned. It may be one of the following values:

Value	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.

IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapOpenMessageEx** function uses the FETCH command to access the specified message on the server. The client can then use the **ImapRead** function to read the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCloseMessage, ImapGetMessage, ImapGetMessageHeaders, ImapOpenMessage, ImapRead, ImapStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapRead Function

```
INT WINAPI ImapRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **ImapRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that there is no more data available to be read. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When **ImapRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetMessage, ImapGetMessageHeaders, ImapIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapRegisterEvent Function

```
INT WINAPI ImapRegisterEvent(
   HCLIENT hClient,
   UINT nEvent,
   IMAPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **ImapRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
IMAP_EVENT_CONNECT	The connection to the server has completed.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **ImapEventProc** callback

function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapRegisterEvent** function associates a callback function with a specific event. The event handler is an **ImapEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a message is being retrieved. The IMAP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one IMAP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDisableEvents, ImapEnableEvents, ImapEventProc, ImapFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapRenameMailbox Function

```
INT WINAPI ImapRenameMailbox(
   HCLIENT hClient,
   LPCTSTR lpszOldMailbox,
   LPCTSTR lpszNewMailbox
);
```

The **ImapRenameMailbox** function renames an existing mailbox.

Parameters

hClient

Handle to the client session.

lpszOldMailbox

A null-terminated string which specifies the mailbox to be renamed.

lpszNewMailbox

A null-terminated string which specifies the new mailbox name.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapRenameMailbox** function renames an existing mailbox on the server. The new mailbox name cannot exist on the server, or the function will fail.

If the existing mailbox name contains inferior hierarchical names (mailboxes under the specified mailbox) then those mailboxes will also be renamed. For example, if the mailbox "Mail/Pictures" contains two mailboxes, "Personal" and "Work" and it is renamed to "Mail/Images" then the two mailboxes under it would be automatically renamed to "Mail/Images/Personal" and "Mail/Images/Work".

If the mailbox being renamed is the currently selected mailbox, the current mailbox will be unselected and any messages marked for deletion will be expunged. The new mailbox name will then automatically be re-selected. To prevent deleted messages from being removed from the mailbox prior to being renamed, use the **ImapUnselectMailbox** function to unselect the current mailbox before calling **ImapRenameMailbox**. Note that if the rename operation fails, the client may be left in an unselected state.

It is permitted to rename the special mailbox INBOX. In this case, the messages will be moved from the INBOX mailbox to the new mailbox. If the INBOX mailbox is currently selected, the new mailbox will not automatically be selected. INBOX will remain the selected mailbox.

If you need to delete a mailbox which contains Unicode characters or symbols, it is recommended you always use the Unicode version of this function. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. The ANSI version of this function will accept mailbox names using either modified UTF-7 encoding or UTF-8 encoding.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapCreateMailbox, ImapDeleteMailbox, ImapGetFirstMailbox, ImapGetNextMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapReselectMailbox Function

```
BOOL WINAPI ImapReselectMailbox(
   HCLIENT hClient,
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ImapReselectMailbox** function reselects the current mailbox and returns updated information about the status of the mailbox.

Parameters

hClient

Handle to the client session.

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains updated information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapReselectMailbox** function forces the current mailbox to be reselected and returns updated information about the status of the mailbox. Deleted messages are not expunged from the mailbox and remain marked for deletion.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapExamineMailbox, ImapSelectMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapSearchMailbox Function

```
LONG WINAPI ImapSearchMailbox(
   HCLIENT hClient,
   LPCTSTR lpszCriteria,
   LPCTSTR lpszCharacterSet,
   DWORD dwReserved,
   UINT *lpnMessageIds,
   LONG nMaxMessages
);
```

The **ImapSearchMailbox** function searches the current mailbox for messages that match the specified criteria, returning matching message identifiers.

Parameters

hClient

Handle to the client session.

lpszCriteria

A null-terminated string which specifies the search criteria.

lpszCharacterSet

A null-terminated string which specifies the character set to use when searching the mailbox. If this parameter is NULL or an empty string, the default UTF-8 character set will be used.

dwReserved

A reserved parameter which should be set to the value 0.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which match the search criteria in the current mailbox. This parameter may be NULL, in which case the function will return the number of matching messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the ***lpnMessageIds*** array. If the ***lpnMessageIds*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of messages that meet the search criteria in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapSearchMailbox** function is used to search a mailbox for messages which match a given criteria and return a list of the matching message identifiers. The search criteria is composed of one or more search keywords and and optional value to match against. String searches are not case sensitive and partial matches in the message are returned.

The following search keywords are recognized:

Keyword	Description
ANSWERED	Match those messages which have the IMAP_FLAG_ANSWERED flag set.

BCC *address*	Match those messages which contain the specified address in the BCC header field.
BEFORE *date*	Match those messages which were added to the mailbox prior to the specified date.
BODY *string*	Match those messages where the body contains the specified string.
CC *address*	Match those messages which contain the specified address in the CC header field.
DELETED	Match those messages which have the IMAP_FLAG_DELETED flag set.
DRAFT	Match those messages which have the IMAP_FLAG_DRAFT flag set.
FLAGGED	Match those messages which have the IMAP_FLAG_URGENT flag set.
FROM *address*	Match those messages which contain the specified address in the FROM header field.
HEADER *field string*	Match those messages which contain the string in the specified header field. If no string is specified, then all messages which contain the header will be matched.
LARGER *size*	Match those messages which are larger than the specified size in bytes.
NEW	Match those messages which have the IMAP_FLAG_RECENT flag set, but not the IMAP_FLAG_SEEN flag.
OLD	Match those messages which do not have the IMAP_FLAG_RECENT flag set.
ON *date*	Match those messages which were added on the specified date.
RECENT	Match those messages which have the IMAP_FLAG_RECENT flag set.
SEEN	Match those messages which have the IMAP_FLAG_SEEN flag set.
SENTBEFORE *date*	Match those messages whose Date header value is earlier than the specified date.
SENTON *date*	Match those messages whose Date header value is the same as the specified date.
SENTSINCE *date*	Match those messages whose Date header value is later than the specified date.
SINCE *date*	Match those messages added to the mailbox after the specified date.
SMALLER *size*	Match those messages which are smaller than the specified size in bytes.
SUBJECT *string*	Match those messages whose Subject header contains the specified string.
TEXT *string*	Match those messages whose headers or body contains the specified string.
TO *address*	Match those messages which contain the specified address in the TO header field.
UID *sequence*	Match those messages with unique identifiers in the sequence set.
UNANSWERED	Match those messages which do not have the IMAP_FLAG_ANSWERED

	flag set.
UNDELETED	Match those messages which do not have the IMAP_FLAG_DELETED flag set.
UNDRAFT	Match those messages which do not have the IMAP_FLAG_DRAFT flag set.
UNFLAGGED	Match those messages which do not have the IMAP_FLAG_URGENT flag set.
UNSEEN	Match those messages which do not have the IMAP_FLAG_SEEN flag set.

In addition to the listed keywords, the keyword NOT may prefix a keyword to return those messages which do not match the search criteria. For example, "NOT TO user@domain.com" would return those messages which were not addressed to user@domain.com.

If multiple search keywords are specified, the result is the intersection of all those messages which meet the search criteria. For example, a search criteria of "DELETED SINCE 1-Jan-2003" would return all those messages which are marked for deletion and were added to the mailbox after 1 January 2003.

Those search keywords which expect dates must be specified in format ***dd-mmm-yyyy*** where the month is the three letter abbreviation for the month name. Note that the internal date the message was added to the mailbox is not the same as the value of the Date header field in the message.

The UID keyword expects a one or more unique message identifiers. These values may provided as comma separated list, or a range delimited by a colon. For example, "UID 23000:24000" would return all those messages who have UIDs ranging from 23000 through to 24000.

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetDeletedMessages, ImapGetMessageCount, ImapGetMessageFlags, ImapGetNewMessages, ImapGetUnseenMessages

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapSelectMailbox Function

```
INT WINAPI ImapSelectMailbox(
   HCLIENT hClient,
   LPCTSTR lpszMailbox,
   LPIMAPMAILBOX lpMailboxInfo
);
```

The **ImapSelectMailbox** function selects the specified mailbox for read-write access.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the new mailbox to be selected. If this parameter is NULL or an empty string, the default INBOX mailbox will be selected.

lpMailboxInfo

A pointer to an IMAPMAILBOX structure which contains information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapSelectMailbox** function is used to select a mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **ImapUnselectMailbox** function prior to selecting the new mailbox.

If you need to select a mailbox which contains Unicode characters or symbols, it is recommended you use the Unicode version of this function. The mailbox name will automatically be converted to modified UTF-7 encoding as defined by the IMAP4 standard. The ANSI version of this function will only accept mailbox names using modified UTF-7 encoding or UTF-8 encoding.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **ImapSelectMailbox** again with the same mailbox name. Note that this will not cause any messages marked for deletion to be expunged.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-only mode, use the **ImapExamineMailbox** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

ImapDeleteMailbox, ImapExamineMailbox, ImapGetFirstMailbox, ImapGetMailboxStatus, ImapGetNextMailbox, ImapRenameMailbox, ImapReselectMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapSetLastError Function

```
VOID WINAPI ImapSetLastError(
   DWORD dwErrorCode
);
```

The **ImapSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the caller. A value of zero clears the last error code.

Return Value

None.

Remarks

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those functions which call **ImapSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **ImapGetLastError** function. The use of **ImapGetLastError** is optional. An application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapGetErrorString, ImapGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapSetMessageFlags Function

```
INT WINAPI ImapSetMessageFlags(
   HCLIENT hClient,
   UINT nMessageId,
   UINT nMode
   DWORD dwMessageFlags
);
```

The **ImapSetMessageFlags** function returns the message flags for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

nMode

An unsigned integer value which specifies one of the following modes which determines how the message flags are set:

Value	Description
IMAP_FLAGS_REPLACE	All message flags are replaced with the flags specified by the ***dwMessageFlags*** parameter.
IMAP_FLAGS_ADD	The message flags specified by the ***dwMessageFlags*** parameter will be set for the message. Message flags that have been previously set will remain unmodified.
IMAP_FLAGS_REMOVE	The message flags specified by the ***dwMessageFlags*** parameter will be removed from the message. Message flags that are not specified will remain unmodified.

dwMessageFlags

An unsigned integer value which specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the function succeeds, it returns IMAP_RESULT_OK. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDeleteMessage, ImapGetMessageCount, ImapGetMessageFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapSetTimeout Function

```
INT WINAPI ImapSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **ImapSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The timeout value is only used with blocking network operations.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapConnect, ImapGetTimeout, ImapRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapStoreMessage Function

```
INT WINAPI ImapStoreMessage(
   HCLIENT hClient,
   UINT nMessageId,
   LPCTSTR lpszFileName
);
```

The **ImapStoreMessage** function retrieves a message from the current mailbox and stores it in a local file or the system clipboard.

Parameters

hClient

Handle to the client session.

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapStoreMessage** function provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

If event handling is enabled, the IMAP_EVENT_PROGRESS event will fire periodically during the transfer of the message to the local system. An application can determine how much of the message has been retrieved by calling the **ImapGetTransferStatus** function.

To retrieve the message into a global memory buffer so that it can be passed to the MIME or SMTP libraries, use the **ImapGetMessage** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapGetMessage, ImapGetMessageHeaders, ImapGetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapSubscribeMailbox Function

```
INT WINAPI ImapSubscribeMailbox(
   HCLIENT hClient,
   LPCTSTR lpszMailbox
);
```

The **ImapSubscribeMailbox** function subscribes the user to the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the mailbox to subscribe to.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapSubscribeMailbox** function adds the specified mailbox to the current user's list of active or subscribed mailboxes. The user will remain subscribed to the mailbox across multiple sessions, until the **ImapUnsubscribeMailbox** function is called to remove the mailbox from the subscription list.

To list those mailboxes which the user has subscribed to, use the **ImapGetFirstMailbox** function and specify the IMAP_LIST_SUBSCRIBED option.

Note that if a user subscribes to a mailbox and that mailbox is later renamed or deleted, the mailbox will not be automatically removed from the user's subscription list. An application must not assume that because a mailbox name is included in the list of subscribed mailboxes, it exists and can be selected. To check if the mailbox exists, use the **ImapGetMailboxStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapExamineMailbox, ImapGetFirstMailbox, ImapGetNextMailbox, ImapSelectMailbox, ImapUnselectMailbox, ImapUnsubscribeMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapUndeleteMessage Function

```
INT WINAPI ImapUndeleteMessage(
    HCLIENT hClient,
    UINT nMessageId
);
```

The **ImapUndeleteMessage** function removes the deletion flag for the specified message.

Parameters

hClient

Handle to the client session.

nMessage

Number of message to undelete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDeleteMessage, ImapGetDeletedMessages, ImapGetMessage, ImapGetMessageCount, ImapGetMessageFlags, ImapReselectMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapUninitialize Function

```
VOID WINAPI ImapUninitialize();
```

The **ImapUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **ImapInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **ImapUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **ImapUninitialize** for every successful call to **ImapInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapConnect, ImapDisconnect, ImapInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapUnselectMailbox Function

```
INT WINAPI ImapUnselectMailbox(
    HCLIENT hClient,
    BOOL bExpunge
);
```

The **ImapUnselectMailbox** function unselects the current mailbox.

Parameters

hClient

Handle to the client session.

bExpunge

A boolean flag which determines if deleted messages will be expunged from the mailbox. A non-zero value specifies that messages that have been marked for deletion will be removed from the mailbox. A zero value specifies that no messages will be removed from the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapDeleteMailbox, ImapExamineMailbox, ImapGetFirstMailbox, ImapGetMailboxStatus, ImapGetNextMailbox, ImapRenameMailbox, ImapReselectMailbox, ImapSelectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapUnsubscribeMailbox Function

```
INT WINAPI ImapUnsubscribeMailbox(
    HCLIENT hClient,
    LPCTSTR lpszMailbox
);
```

The **ImapUnsubscribeMailbox** function unsubscribes the user from the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A null-terminated string which specifies the mailbox to unsubscribe from.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapUnsubscribeMailbox** function removes the specified mailbox from the current user's list of active or subscribed mailboxes.

To list those mailboxes which the user has subscribed to, use the **ImapGetFirstMailbox** function and specify the IMAP_LIST_SUBSCRIBED option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ImapExamineMailbox, ImapGetFirstMailbox, ImapGetNextMailbox, ImapSelectMailbox, ImapSubscribeMailbox, ImapUnselectMailbox

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ImapWrite Function

```
INT WINAPI ImapWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **ImapWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the IMAP_EVENT_WRITE asynchronous notification message before it resumes sending data.

The **ImapWrite** function should not be used to send commands to the IMAP server. Use the **ImapCommand** function instead.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

See Also

ImapCommand, ImapIsBlocking, ImapIsWritable, ImapRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Internet Message Access Protocol Data Structures

- IMAPMAILBOX
- IMAPMAILBOXSTATUS
- IMAPMESSAGE
- IMAPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPMAILBOX

This structure contains information about a selected mailbox.

```
typedef struct _IMAPMAILBOX
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessageId;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwFlags;
    DWORD   dwPermanentFlags;
    DWORD   dwAccessMode;
    DWORD   dwReserved1;
    DWORD   dwReserved2;
} IMAPMAILBOX, *LPIMAPMAILBOX;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessageId

A value which specifies the message ID of the first unseen message in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwFlags

A value which specifies one or more mailbox flags. One or more of the following values may be specified:

Value	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any child mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this

	flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

dwPermanentFlags

A value which specifies the message flags that a client can change permanently. If this value is zero, then no permanent flags are defined for the mailbox and the client may assume that all message flags may be set permanently. Otherwise, one or more of the following values may be specified:

Value	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

dwAccessMode

A value which specifies the access mode for the mailbox. It may be one of the following values:

Value	Description
IMAP_ACCESS_READONLY	The mailbox has been selected in read-only mode. Messages may not be created in the mailbox, nor can message flags be modified.
IMAP_ACCESS_READWRITE	The mailbox has been selected in read-write mode. Messages may be modified by the client, and messages marked for deletion can be expunged.

dwReserved1

A reserved value that is undefined.

dwReserved2

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPMAILBOXSTATUS

This structure contains information about a mailbox.

```
typedef struct _IMAPMAILBOXSTATUS
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessages;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwReserved;
} IMAPMAILBOXSTATUS, *LPIMAPMAILBOXSTATUS;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessages

A value which specifies the number of unread messages in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPMESSAGE

This structure contains information about a message.

```
typedef struct _IMAPMESSAGE
{
    UINT    nMessageId;
    DWORD   dwMessageUID;
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwTimestamp;
    DWORD   dwReserved;
} IMAPMESSAGE, *LPIMAPMESSAGE;
```

Members

nMessageId

A sequential integer value which identifies the message. The message identifier is only valid while the mailbox is selected and no messages marked for deletion have been expunged. To maintain a persistent identifier for the message, use a combination of the mailbox UID and message UID.

dwMessageUID

An integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque. If the client application stores the message UID on the local system, it should also store the UID for the mailbox that contains the message. If the mailbox UID changes, the message UID may no longer be valid.

dwSize

Specifies the size of the message in bytes. This value may not be the same as the number of characters in the message, particularly if the message text uses Unicode or extended multibyte ANSI sequences. For a structured multipart MIME message, this includes the combined size of all message parts.

dwFlags

A value which specifies one or more message flags. One or more of the following values may be specified:

Value	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

dwTimestamp

An integer value which specifies the date and time that the message was added to the mailbox. The value is expressed as the number of seconds since midnight, 1 January 1970 and is the same value that is used for the standard C runtime library time methods. Note that the date and time used is

the message's internal date from the mail server, not the value of the **Date** header field in the message.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmapv12.lib

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

IMAPTRANSFERSTATUS

This structure is used by the ImapGetTransferStatus function to return information about a file transfer in progress.

```
typedef struct _IMAPTRANSFERSTATUS
{
   UINT    nMessageId;
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} IMAPTRANSFERSTATUS, *LPIMAPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Library

The Large Language Model API provides a unified, high-level interface for integrating large language model capabilities into applications. It simplifies working with different providers by handling connections, messaging, and conversation state internally, while still giving developers control over configuration, performance, and behavior. The API is designed to support both simple use cases and more advanced scenarios involving model selection, token management, and session control.

Reference

- Functions
- Constants
- Data Structures
- Providers
- Error Codes

Library Information

File Name	CSLLMV12.DLL
Version	12.0.3024.1286
LibID	E77A6A0A-3B86-49D9-9A3D-A0B425BDA66D
Import Library	CSLLMV12.LIB
Dependencies	None

Overview

The SocketTools LLM API is built around a session-based architecture that encapsulates all aspects of interacting with a language model, including connection details, configuration settings, and conversation history. This design allows developers to treat each interaction as part of a managed context, making it easier to maintain state across multiple requests without needing to manually track prior messages or model parameters. Sessions can be configured with defaults or customized with options such as system prompts, token limits, and behavioral settings, depending on the needs of the application.

A central goal of the API is to abstract the differences between providers and models, presenting a unified interface that works consistently regardless of the underlying service. Developers can discover available providers and models, query their capabilities, and select appropriate configurations at runtime. This flexibility makes it possible to switch between services or adapt to new models with minimal code changes, while still exposing enough detail to fine-tune behavior when needed.

The API also provides robust support for managing conversation data. It maintains a structured history of messages and responses, enabling applications to iterate through prior interactions, search for specific content, or limit the amount of retained context to control memory usage and token consumption. This is particularly important when working with models that have strict context limits, as it allows developers to balance response quality with performance and cost considerations.

In addition to core messaging functionality, the API includes a range of utility features that help developers monitor and optimize their applications. These include estimating and tracking token usage, managing cache lifetimes for model metadata, handling errors in a consistent way, and optionally logging network activity for debugging purposes. Together, these capabilities provide

insight into how the application is interacting with the model and help identify potential performance or configuration issues.

Overall, the LLM API is designed to provide a practical balance between simplicity and control. It enables developers to quickly add AI-powered features to their applications while retaining the flexibility to customize behavior, manage resources, and integrate with different providers as requirements evolve.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Functions

Function	Description
LlmAskModel	Sends a prompt to the specified service provider and model.
LlmAttachThread	Attaches the specified client handle to another thread.
LlmCancelRequest	Cancels the current blocking request.
LlmClearHeaders	Clears the custom HTTP request headers for the client session.
LlmClearHistory	Clears the current conversation history for a client.
LlmConnect	Establishes a connection using the specified provider, URL, model, API key, and options.
LlmConnectEx	Establishes a connection using an existing session structure.
LlmConvertTimestamp	Converts a message or response timestamp value into a SYSTEMTIME structure.
LlmCreateSession	Creates a session object using the specified provider settings.
LlmCreateSessionEx	Creates a session object with extended configuration such as prompt, token limit, and sampling settings.
LlmDeleteSession	Deletes a session object and releases its resources.
LlmDisableTrace	Disables logging of network function calls.
LlmDisconnect	Disconnects from the provider and releases the session handle.
LlmEnableTrace	Enables logging of network function calls to a file.
LlmEnumModels	Enumerates the models available for the connected provider.
LlmEnumProviders	Enumerates the supported LLM providers.
LlmEstimateTokenCount	Estimates the number of tokens in a text string for a specific provider.
LlmExportHistory	Exports the current conversation history to a JSON-formatted file.
LlmFindMessage	Searches the conversation history for a message matching the specified text or flags.
LlmFormatTimestamp	Formats a message timestamp as an ISO 8601 date and time string.
LlmGetCacheTime	Returns the current model or metadata cache lifetime for a client.
LlmGetClientOptions	Returns the current client option flags.
LlmGetClientProvider	Returns the provider associated with a client, and optionally its base URL.
LlmGetDefaultModelFlags	Returns the default capability flags for the specified model.
LlmGetDefaultModelName	Returns the default model name for the specified provider.
LlmGetErrorString	Returns a text description for the specified error code.
LlmGetFirstMessage	Retrieves the first message in the conversation history.
LlmGetIdleTime	Returns the current idle timeout period for a client session.
LlmGetHistorySize	Returns the maximum number of messages which can be stored in the conversation history.
LlmGetLastError	Returns the last error code set for the current thread.
LlmGetMemoryUsage	Returns the amount of memory currently used by a client session.
LlmGetMessage	Retrieves a specific message prompt and response using the history index.
LlmGetMessageById	Retrieves a specific message prompt and response using a message ID.
LlmGetMessageCount	Returns the current number of messages in the conversation history.
LlmGetModelContextSize	Returns the context window size for the specified model.
LlmGetModelFlags	Returns the current model behavior flags for a client.
LlmGetModelInfo	Returns detailed information about a specific model.
LlmGetModelName	Returns the name of the current model used with a client session.
LlmGetNextMessage	Retrieves the next message in the conversation history.

LlmGetProviderFlags	Returns the capability flags associated with a provider.
LlmGetProviderId	Returns a provider ID based on a provider name, endpoint URL or model name.
LlmGetProviderInfo	Retrieves detailed information about a provider.
LlmGetProviderName	Returns the display name of a provider.
LlmGetProviderUrl	Returns the base endpoint URL for the specified provider.
LlmGetResponseHeader	Returns the value of an HTTP header from the server response.
LLmGetStatusCode	Returns the most recent HTTP status code for a client request.
LlmGetSystemPrompt	Returns the current system prompt for a client session.
LlmGetTimeout	Returns the number of seconds until a request times out.
LlmGetTokenUsage	Returns token usage statistics for the client session.
LlmGetUserAgent	Returns the current user-agent string used by the client.
LlmImportHistory	Imports a conversation history from the specified JSON-formatted file.
LlmInitialize	Initializes the library and validates the specified license key at runtime.
LlmIsBlocking	Returns whether the client is currently processing a blocking operation.
LlmIsConnected	Returns whether the client session is valid and connected to a server.
LlmNormalizeText	Normalizes message text into a consistent format.
LlmResetSession	Resets the current client session to its initial state.
LlmSendMessage	Sends a prompt to the model and returns the response as a string.
LlmSendMessageEx	Sends a structured message to the model and returns a structured response.
LlmSetCacheTime	Sets the cache lifetime used by the client for model or provider data.
LlmSetClientOptions	Sets the current client option flags.
LlmSetHistorySize	Sets the maximum number of messages retained in the conversation history.
LlmSetIdleTime	Sets the idle timeout period for a client session.
LlmSetLastError	Sets the last error code for the current thread.
LlmSetModelFlags	Sets model behavior flags for a client session.
LlmSetModelName	Changes the current model for a client session.
LlmSetRequestHeader	Sets the value of a custom HTTP header issued with subsequent requests.
LlmSetSystemPrompt	Sets the system prompt for a client session.
LlmSetTimeout	Sets the number of seconds until a blocking operation times out.
LlmSetUserAgent	Sets the user-agent string used by the client.
LlmTrimHistory	Removes messages from the conversation history, optionally constrained by token usage.
LlmUninitialize	Terminates use of the library by the calling process.
LlmValidateModel	Validates that a model name is recognized or supported by the current provider.
LlmValidateText	Performs basic validation on message text before it is submitted to a model.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmAskModel Function

```
BOOL WINAPI LlmAskModel (
   DWORD dwProviderId,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey,
   LPCTSTR lpszPrompt,
   LPTSTR lpszResponse,
   INT nMaxLength
);
```

The **LlmAskModel** function sends a single prompt to a model and returns the generated response without requiring the application to manage a client session.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. See the remarks below for more information about service providers.

lpszBaseUrl

A pointer to a null-terminated string that specifies the base URL for the provider endpoint. This parameter may be NULL to use the default endpoint for the selected provider. Some providers, such as Microsoft Foundry or local services, require a specific base URL.

lpszModelName

A pointer to a null-terminated string that specifies the model name to use for the session. This parameter may be NULL to use the provider default model, if one is defined. Providers without a default model require an explicit model name. This typically includes models accessed using the Microsoft Foundry or locally hosted inference providers.

lpszApiKey

A pointer to a null-terminated string that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries.

lpszPrompt

A pointer to a null-terminated string which specifies the prompt which should be sent to the model. This parameter cannot be NULL or a zero-length string. If the string consists only of whitespace characters, or contains invalid Unicode or non-text data, this function will fail.

lpszResponse

A pointer to a string buffer which will contain the model's response to the prompt. The response will always be null terminated and if the buffer is not large enough to store the complete response, the response will be truncated. When the function is called, this buffer is always initialized to an empty string and will contain the response if the function returns successfully.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the response buffer, including the terminating null character. If the length is an invalid value, such as zero or a negative number, the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmAskModel** function provides a simplified way to send a single prompt to a model and return the generated response. It does not require a client handle. It creates a connection using the default options and timeout period, sends the request and then closes the connection. This function manages the session automatically and is intended for simple request and response operations.

This function does not preserve conversation history between calls. Applications which require conversational context, persistent connections or advanced session management should create a session using the **LlmConnect** function and then send prompts using the **LlmSendMessage** function.

The *dwProviderId* parameter specifies the numeric identifier assigned to a supported provider. Refer to the providers page for a list of supported service providers and their default models.

If the *lpszBaseUrl* parameter is NULL or a zero-length string, the library will use a provider-specific default endpoint. Some providers, such as Microsoft Foundry services or locally hosted models, require an explicit endpoint and will fail if one is not provided.

If the *lpszModelName* parameter is NULL or a zero-length string, the library will attempt to use a default model for the selected provider. Not all providers define a default model, and in those cases an explicit model name must be provided when establishing the connection. This typically includes models accessed using the Microsoft Foundry or locally hosted inference providers.

If the service provider requires authentication, the API key can be provided in several formats, allowing applications to provide credentials in a way that balances convenience and security:

- **Literal string**
  The API key is provided directly as a string. This is the simplest approach and is convenient for testing or small internal tools, but it is not recommended for production applications. A literal key may be exposed in source code, configuration files, diagnostic logs or the compiled application binary.
- **Environment variable**
  The name of an environment variable enclosed in percent symbols, for example `"%OPENAI_API_KEY%"`. The value of the environment variable is used as the API key. This avoids embedding the key directly in the application and is useful for development, scripts and deployment environments. However, environment variables may still be visible to the current user, inherited by child processes or exposed through system configuration, so they should still be protected appropriately.
- **Windows credential**
  A credential name enclosed in braces, for example `"{MyAppName/OpenAI}"`. The API key is retrieved from the Windows Credential Manager using the specified name. This is the recommended approach for most Windows applications because the secret is stored outside the application and managed by the operating system. The credential can be created manually using Credential Manager, or programmatically by the application using the Windows Credential Management API. Applications should choose a unique credential name and document it clearly if users or administrators are expected to configure the value themselves.

For production applications, it is recommended that you avoid using literal API key strings and instead use an environment variable or a value stored in the Windows Credential Manager. If you must

provide the API key as a literal string, you should take additional steps to protect it, such as storing the key in an encrypted form and limiting access to authorized users or processes. Avoid embedding API keys directly in application source code, configuration files, installers or other content that could be exposed through version control systems, logs or public repositories.

This function will block the calling thread until the server returns a response or the default timeout period has elapsed. Larger or more complex models may take longer to generate a response. Locally hosted models may also take longer to respond than cloud-based models, depending on the model type and size.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    LPCTSTR lpszPrompt = _T("What is the capital of the state of California?");
    TCHAR szResponse[4096] = _T("");

    // Use a locally hosted model with LM Studio
    BOOL bResult = LlmAskModel(
            LLM_PROVIDER_LOCAL,              // Local provider
            _T("http://127.0.0.1:1234/v1/"), // Localhost endpoint
            _T("nvidia/nemotron-3-nano-4b"), // Model name
            NULL,
            lpszPrompt,
            szResponse,
            _countof(szResponse));

    if (bResult)
    {
        _tprintf(_T("The response was %d characters:\n"), lstrlen(szResponse));
        _putts(szResponse);
    }
    else
    {
        DWORD dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    LlmUninitialize();
    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnect, LlmDisconnect, LlmInitialize, LlmSendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmAttachThread Function

```
DWORD WINAPI LlmAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **LlmAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

When a client handle is created, it is associated with the thread that created it. If another thread attempts to use the handle, the function will fail because that thread does not own the handle. This prevents other threads from interfering with an operation being performed by the owner thread. The **LlmAttachThread** function changes the thread that owns the client handle. This is useful when one thread creates the client session and then passes it to a worker thread for processing. The function returns the thread ID of the previous owner, which can be saved and used later to restore ownership of the handle.

This function should normally be called by the new owner thread immediately after it has been created. If the worker thread does not release the handle itself, ownership should be restored to the original thread before the worker thread terminates. Do not use **LlmAttachThread** to forcibly take ownership of a handle from another thread while a blocking operation is in progress. The handle should only be transferred when no other thread is actively using it.

The ***dwThreadId*** parameter is assumed to be a valid thread ID. This function does not verify that the specified thread exists. Specifying an invalid thread ID will orphan the handle until **LlmUninitialize** is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnect, LlmDisconnect, LlmUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmCancelRequest Function

```
BOOL WINAPI LlmCancelRequest(
    HCLIENT hClient
);
```

The **LlmCancelRequest** function cancels the current blocking request.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmCancelRequest** function cancels the current blocking request by closing the active HTTP connection associated with the client session. This causes the blocking function to fail and return control to the calling thread. The last error code for the blocking function will be set to indicate the operation was canceled.

This function cancels the request locally. It does not guarantee that the provider will stop processing a request that has already been received. The provider may still complete the request and may charge for any processing or tokens used before the connection was closed.

Canceling a request does not delete the client session or clear the conversation history. The client handle remains valid, and a new HTTP connection will be established automatically if it is needed for a subsequent request.

If no blocking request is currently in progress, this function succeeds and has no effect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmIsBlocking, LlmSendMessage, LlmSendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmClearHeaders Function

```
BOOL WINAPI LlmClearHeaders(
    HCLIENT hClient
);
```

The **LlmClearHeaders** function removes all custom HTTP request headers for the client session.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmClearHeaders** function removes all custom request header values set using the **LlmSetRequestHeader** method. This is useful when you want to ensure that custom headers used for a previous call to **LlmSendMessage** are not included with subsequent requests made during the same session.

This method does not clear authentication credentials, organization identifiers, project identifiers, or any required headers automatically generated by the control.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetResponseHeader, LlmResetSession, LlmSendMessage, LlmSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmClearHistory Function

```
BOOL WINAPI LlmClearHistory(
    HCLIENT hClient
);
```

The **LlmClearHistory** function clears the conversation history for the current client session.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmClearHistory** function removes all messages and responses stored in the conversation history for the specified client session. This can be used to begin a new conversation while preserving the current connection, provider settings, model selection, timeout, system prompt and other session options.

This function only clears the local history maintained by the client. It does not reset provider-specific settings, delete cached model information or close the active HTTP connection. It also does not change the current system prompt.

After the history has been cleared, subsequent calls to **LlmSendMessage** or **LlmSendMessageEx** will not include any previous messages from the conversation history.

Clearing the conversation history invalidates any string pointers previously returned in LLM_RESPONSE structures or by message history functions. Applications must not use those pointers after this function returns. If the application needs to keep any message or response text, it should make its own copy before clearing the history.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetHistorySize, LlmSetHistorySize, LlmGetFirstMessage, LlmGetNextMessage, LlmResetSession, LlmTrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmConnect Function

```
HCLIENT WINAPI LlmConnect(
   DWORD dwProviderId,
   DWORD dwOptions,
   INT nTimeout,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey
);
```

The **LlmConnect** function establishes a connection to the specified large language model service provider and creates a client session.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. See the remarks below for more information about service providers.

dwOptions

Specifies one or more option flags that control how the connection is established. This parameter can be zero or a combination of the following values:

Value	Description
LLM_OPTION_NONE	A standard connection to the server is established using the default behavior for the selected provider. No additional options are enabled, and the connection will use the provider's default protocol, caching behavior and request headers.
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be

	used when a provider does not require a User-Agent header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.
LLM_OPTION_DEFAULT	Specifies the default option set, which is equivalent to `LLM_OPTION_NONE`. No additional flags are enabled and the connection uses standard behavior.

nTimeout

The timeout period for connection and request operations, in seconds. A value of zero indicates that the default timeout should be used.

lpszBaseUrl

A pointer to a null-terminated string that specifies the base URL for the provider endpoint. This parameter may be NULL to use the default endpoint for the selected provider. Some providers, such as Microsoft Foundry or local services, require a specific base URL.

lpszModelName

A pointer to a null-terminated string that specifies the model name to use for the session. This parameter may be NULL to use the provider default model, if one is defined. Providers without a default model require an explicit model name. This typically includes models accessed using the Microsoft Foundry or locally hosted inference providers.

lpszApiKey

A pointer to a null-terminated string that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmConnect** function creates a client session that is used for all subsequent operations with the selected provider. The returned handle maintains the connection settings, authentication information, selected model, and conversation history for the session.

The ***dwProviderId*** parameter specifies the numeric identifier assigned to a supported provider. Provider identifiers are predefined constants built into the library and are assigned fixed values beginning at 1 and increasing sequentially for each provider. These identifiers are stable and are intended to uniquely identify a provider regardless of its display name. Applications should use the predefined constants (such as `LLM_PROVIDER_OPENAI` or `LLM_PROVIDER_MICROSOFT`) rather than hard-coding numeric values directly in source code.

Refer to the providers page for a list of supported service providers and their default models.

If the ***lpszBaseUrl*** parmeter is NULL or a zero-length string, the library will use a provider-specific

default endpoint. Some providers, such as Microsoft Foundry services or locally hosted models, require an explicit endpoint and will fail if one is not provided.

If the *lpszModelName* parameter is NULL or a zero-length string, the library will attempt to use a default model for the selected provider. Not all providers define a default model, and in those cases an explicit model name must be provided when establishing the connection. This typically includes models accessed using the Microsoft Foundry or locally hosted inference providers.

If the service provider requires authentication, the API key can be provided in several formats, allowing applications to provide credentials in a way that balances convenience and security:

- **Literal string**
  The API key is provided directly as a string. This is the simplest approach and is convenient for testing or small internal tools, but it is not recommended for production applications. A literal key may be exposed in source code, configuration files, diagnostic logs or the compiled application binary.
- **Environment variable**
  The name of an environment variable enclosed in percent symbols, for example `"%OPENAI_API_KEY%"`. The value of the environment variable is used as the API key. This avoids embedding the key directly in the application and is useful for development, scripts and deployment environments. However, environment variables may still be visible to the current user, inherited by child processes or exposed through system configuration, so they should still be protected appropriately.
- **Windows credential**
  A credential name enclosed in braces, for example `"{MyAppName/OpenAI}"`. The API key is retrieved from the Windows Credential Manager using the specified name. This is the recommended approach for most Windows applications because the secret is stored outside the application and managed by the operating system. The credential can be created manually using Credential Manager, or programmatically by the application using the Windows Credential Management API. Applications should choose a unique credential name and document it clearly if users or administrators are expected to configure the value themselves.

For production applications, it is recommended that you avoid using literal API key strings and instead use an environment variable or a value stored in the Windows Credential Manager. If you must provide the API key as a literal string, you should take additional steps to protect it, such as storing the key in an encrypted form and limiting access to authorized users or processes. Avoid embedding API keys directly in application source code, configuration files, installers or other content that could be exposed through version control systems, logs or public repositories.

This method will block the calling thread until the connection has been established or the timeout period has elapsed. The connection established by this method represents a logical session. Depending on the provider and options specified, the underlying HTTP connection may be reused across requests or recreated as needed. Options such as `LLM_OPTION_KEEPALIVE` can be used to control connection behavior.

The client handle returned by this function is owned by the thread that created it. To transfer ownership to another thread, use the **LlmAttachThread** function.

Example

```
#include "cstools12.h"
#include <stdio.h>
```

```
int _tmain(void)
{
    HCLIENT hClient = INVALID_CLIENT;
    DWORD dwError = NO_ERROR;

    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    // Establish a connection to OpenAI using the default endpoint
    // and a lightweight model
    hClient = LlmConnect(
        LLM_PROVIDER_OPENAI,
        LLM_OPTION_DEFAULT,
        LLM_TIMEOUT,
        NULL,                   // Use default URL
        _T("gpt-5.4-nano"),     // Model name
        _T("%OPENAI_API_KEY%") // Environment variable name
    );

    if (hClient == INVALID_CLIENT)
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        LlmUninitialize();
        return 1;
    }

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse); // number of TCHARs
    DWORD dwMessageId = 0;

    // Send a simple message to the model
    if (LlmSendMessage(hClient,
                       _T("What is the capital city of Denmark?"),
                       szResponse,
                       &dwResponseLength,
                       &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Clean up
    LlmDisconnect(hClient);
    LlmUninitialize();

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnectEx, LlmDisconnect, LlmEnumModels, LlmGetProviderInfo, LlmInitialize, LlmIsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmConnectEx Function

```
HCLIENT WINAPI LlmConnectEx(
   DWORD dwProviderId,
   DWORD dwOptions,
   LPLLM_SESSION lpSession
);
```

The **LlmConnectEx** function establishes a connection to the specified large language model service provider using an extended session configuration.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. See the remarks below for more information about service providers.

dwOptions

Specifies one or more option flags that control how the connection is established. This parameter can be zero or a combination of **LLM_OPTION** values. These options control behavior such as connection security, proxy usage, and HTTP connection handling.

Value	Description
LLM_OPTION_NONE	A standard connection to the server is established using the default behavior for the selected provider. No additional options are enabled, and the connection will use the provider's default protocol, caching behavior and request headers.
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be used when a provider does not require a User-Agent header.

LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.
LLM_OPTION_DEFAULT	Specifies the default option set, which is equivalent to `LLM_OPTION_NONE`. No additional flags are enabled and the connection uses standard behavior.

lpSession

A pointer to an `LLM_SESSION` structure that specifies extended configuration options for the client session. This structure can be used to define parameters such as the base URL, model name, authentication information, timeout values, token limits and sampling settings.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmConnectEx** function creates a client session using the configuration specified in the ***lpSession*** structure. This provides more control over how the connection is established compared to **LlmConnect**, and is intended for applications that require advanced configuration.

The ***dwProviderId*** parameter specifies the numeric identifier assigned to a supported provider. Provider identifiers are predefined constants built into the library and are assigned fixed values beginning at 1 and increasing sequentially for each provider. These identifiers are stable and are intended to uniquely identify a provider regardless of its display name. Applications should use the predefined constants (such as `LLM_PROVIDER_OPENAI` or `LLM_PROVIDER_MICROSOFT`) rather than hard-coding numeric values directly in source code.

Refer to the providers page for a list of supported service providers and their default models.

The ***lpSession*** structure allows the application to specify connection parameters such as the provider endpoint, model name, API key, timeout values, token limits and sampling options. Any values not explicitly specified will use provider defaults.

Applications using languages other than C or C++ may find it easier to create and manage session structures using the **LlmCreateSession**, **LlmCreateSessionEx** and **LlmDeleteSession** functions. These helper functions allocate and initialize a compatible structure that can be passed to this function.

The library makes an internal copy of the string members in the LLM_SESSION structure when the connection is established. The application does not need to retain them after the call to **LlmConnectEx** returns.

This function will block the calling thread until the connection has been established or the timeout period has elapsed. Applications should avoid calling this function on the main user interface thread. For best results, create a worker thread to establish the connection, especially when multiple connections are required.

The connection established by this function represents a logical session. Depending on the provider and options specified, the underlying HTTP connection may be reused across requests or recreated as

needed. Options such as `LLM_OPTION_KEEPALIVE` can be used to control connection behavior.

The client handle returned by this function is owned by the thread that created it. To transfer ownership to another thread, use the **LlmAttachThread** function.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    HCLIENT hClient = INVALID_CLIENT;
    DWORD dwError = NO_ERROR;

    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    // Additional options such as temperature, token limits and timeouts
    // can be configured using members of the LLM_SESSION structure
    LLM_SESSION sessionInfo;

    ZeroMemory(&sessionInfo, sizeof(sessionInfo));
    sessionInfo.dwSize = sizeof(sessionInfo);
    sessionInfo.lpszModelName = _T("gpt-5.4-nano");
    sessionInfo.lpszApiKey = _T("%OPENAI_API_KEY%");
    sessionInfo.lpszSystemPrompt = _T("You are a helpful assistant.");

    // Establish a connection to OpenAI using a lightweight model
    // with the default connection options
    hClient = LlmConnectEx(LLM_PROVIDER_OPENAI,
                           LLM_OPTION_DEFAULT,
                           &sessionInfo);

    if (hClient == INVALID_CLIENT)
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        LlmUninitialize();
        return 1;
    }

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse); // number of TCHARs
    DWORD dwMessageId = 0;

    // Send a simple message to the model
    if (LlmSendMessage(hClient,
                       _T("What is the capital city of Denmark?"),
                       szResponse,
                       &dwResponseLength,
                       &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
```

```
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Clean up
    LlmDisconnect(hClient);
    LlmUninitialize();

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnect, LlmCreateSession, LlmCreateSessionEx, LlmDeleteSession, LlmDisconnect, LlmEnumModels, LlmGetProviderInfo, LlmIsConnected, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmConvertTimestamp Function

```
BOOL WINAPI LlmConvertTimestamp(
   TIME_T tTimestamp,
   LPSYSTEMTIME lpSystemTime
   BOOL bLocalTime
);
```

The **LlmConvertTimestamp** function converts a message or response timestamp to a SYSTEMTIME structure.

Parameters

tTimestamp

A 64-bit timestamp value associated with a message or response. This value represents the number of seconds that have elapsed since January 1, 1970 UTC.

lpSystemTime

A pointer to a `SYSTEMTIME` structure that receives the converted date and time value.

bLocalTime

If this parameter is non-zero, the timestamp is converted to the local system time. If this parameter is zero, the timestamp is converted to Coordinated Universal Time (UTC).

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmConvertTimestamp** function converts an internal timestamp value used by the library into a standard `SYSTEMTIME` structure. This allows applications to display message and response times in a human-readable format. These timestamps are used in the client history of requests sent to the model and the responses it returned.

The *tTimestamp* parameter is a standard 64-bit Unix time value and can be used interchangeably with functions in the C standard library such as **gmtime** or **localtime**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps are returned by functions such as **LlmSendMessage** and can be converted into a local date and time using the **LlmConvertTimestamp** function.

The converted time can be returned as either local time or UTC by specifying the *bLocalTime* parameter. Applications should use local time for display purposes and UTC when storing or comparing timestamps across systems. The SYSTEMTIME value can be formatted using standard Windows functions such as **GetDateFormatEx** and **GetTimeFormatEx** to display locale-specific date and time values.

If you wish to convert the message timestamp to a standardized ISO 8601 string value, use the **LlmFormatTimestamp** function.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };
```

```
if (LlmGetFirstMessage(hClient, 0, &clientMessage, &serverResponse))
{
    do
    {
        SYSTEMTIME stMessage = {0};
        TCHAR szDate[64] = {0};
        TCHAR szTime[64] = {0};

        LlmConvertTimestamp(clientMessage.tTimestamp, &stMessage, TRUE);

        GetDateFormat(LOCALE_USER_DEFAULT, 0,
                      &stMessage, NULL,
                      szDate, _countof(szDate));

        GetTimeFormat(LOCALE_USER_DEFAULT, 0,
                      &stMessage, NULL,
                      szTime, _countof(szTime));

        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Timestamp:       %s %s\n"), szDate, szTime);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        SYSTEMTIME stResponse = {0};
        LlmConvertTimestamp(serverResponse.tTimestamp, &stResponse, TRUE);

        GetDateFormat(LOCALE_USER_DEFAULT, 0,
                      &stResponse, NULL,
                      szDate, _countof(szDate));

        GetTimeFormat(LOCALE_USER_DEFAULT, 0,
                      &stResponse, NULL,
                      szTime, _countof(szTime));

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Timestamp:       %s %s\n"), szDate, szTime);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (LlmGetNextMessage(hClient, &clientMessage, &serverResponse));
}
```

This example uses the **GetDateFormat** and **GetTimeFormat** functions for simplicity and compatibility with both ANSI and Unicode builds. On current Windows platforms, applications should prefer using **GetDateFormatEx** and **GetTimeFormatEx**, which provide improved locale support and are recommended for new development. Note that these newer functions are implemented as Unicode-only APIs.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetFirstMessage, LlmFormatTimestamp, LlmGetNextMessage, LlmSendMessageEx, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmCreateSession Function

```
BOOL WINAPI LlmCreateSession(
   DWORD dwProviderId,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey,
   LPCTSTR lpszApiVersion,
   LPLLM_SESSION *lppSession
);
```

The **LlmCreateSession** function allocates and initializes an `LLM_SESSION` structure using the specified connection parameters.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the `LLM_PROVIDER` constants. Refer to the providers page for a list of supported providers and their default models.

lpszBaseUrl

A pointer to a null-terminated string that specifies the base URL for the provider endpoint. This parameter may be NULL to use the default endpoint for the selected provider. Some providers, such as Microsoft Foundry or local services, require a specific base URL.

lpszModelName

A pointer to a null-terminated string that specifies the model name to use for the session. This parameter may be NULL to use the provider default model, if one is defined. Providers without a default model require an explicit model name.

lpszApiKey

A pointer to a null-terminated string that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries.

lpszApiVersion

A pointer to a null-terminated string that specifies the API version required by the provider. This parameter may be NULL if the provider does not require an explicit version. Some providers, such as Microsoft and Anthropic, may require a version string when establishing a connection.

lppSession

A pointer to a variable that receives a pointer to an allocated `LLM_SESSION` structure. On successful return, this parameter will contain a pointer to a structure that can be passed to **LlmConnectEx**. The structure must be released using the **LlmDeleteSession** function when it is no longer needed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmCreateSession** function is a helper function that allocates and initializes an `LLM_SESSION` structure using the specified parameters. This simplifies working with session structures, particularly in

languages that do not easily support structure initialization or require explicit memory management. This function is typically used in conjunction with **LlmConnectEx** when an application requires extended session configuration.

The structure returned by this function is allocated by the library and must be released using the **LlmDeleteSession** function. Failing to release the structure will result in a memory leak.

Applications using languages that do not support pointer-to-pointer parameters may treat the *lppSession* parameter as an opaque pointer value. In this case, the variable should be defined using an integer type that is large enough to hold a pointer value, such as UINT_PTR or an equivalent type. Applications must ensure that the correct pointer size is used for the target platform.

Once a connection has been established using **LlmConnectEx**, the session structure is no longer required and may be released using **LlmDeleteSession**. The library creates an internal copy of the session configuration, so the original structure does not need to remain allocated for the lifetime of the client session.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    HCLIENT hClient = INVALID_CLIENT;
    LPLLM_SESSION lpSession = NULL;
    DWORD dwError = NO_ERROR;

    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    // Create a session structure using helper function
    if (!LlmCreateSession(LLM_PROVIDER_OPENAI,
                          NULL,                      // Default URL
                          _T("gpt-5.4-nano"),        // Model name
                          _T("%OPENAI_API_KEY%"),    // API key
                          NULL,                      // API version
                          &lpSession))
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to create session, error %lu\n"), dwError);
        LlmUninitialize();
        return 1;
    }

    // Establish a connection using the session structure
    hClient = LlmConnectEx(LLM_PROVIDER_OPENAI,
                           LLM_OPTION_DEFAULT,
                           lpSession);

    if (hClient == INVALID_CLIENT)
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
```

```
        // Clean up session before exit
        LlmDeleteSession(&lpSession);
        LlmUninitialize();
        return 1;
    }

    // The session structure is no longer needed after connecting
    LlmDeleteSession(&lpSession);

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse);
    DWORD dwMessageId = 0;

    if (LlmSendMessage(hClient,
                       _T("What is the capital city of Denmark?"),
                       szResponse,
                       &dwResponseLength,
                       &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    LlmDisconnect(hClient);
    LlmUninitialize();

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnectEx, LlmCreateSessionEx, LlmDeleteSession, LlmSendMessage, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmCreateSessionEx Function

```
BOOL WINAPI LlmCreateSessionEx(
   DWORD dwProviderId,
   DWORD dwReserved,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName,
   LPCTSTR lpszApiKey,
   LPCTSTR lpszApiVersion,
   LPCTSTR lpszSystemPrompt,
   LONG nMaxTokens,
   FLOAT fTemperature,
   FLOAT fTopP,
   LPLLM_SESSION *lppSession
);
```

The **LlmCreateSessionEx** function allocates and initializes an `LLM_SESSION` structure using the specified session parameters.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the `LLM_PROVIDER` constants. Refer to the providers page for a list of supported providers and their default models.

dwReserved

Reserved for future use. This parameter must be zero.

lpszBaseUrl

A pointer to a null-terminated string that specifies the base URL for the provider endpoint. This parameter may be NULL to use the default endpoint for the selected provider. Some providers, such as Microsoft Foundry or local services, require a specific base URL.

lpszModelName

A pointer to a null-terminated string that specifies the model name to use for the session. This parameter may be NULL to use the provider default model, if one is defined. Providers without a default model require an explicit model name.

lpszApiKey

A pointer to a null-terminated string that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries.

lpszApiVersion

A pointer to a null-terminated string that specifies the API version required by the provider. This parameter may be NULL if the provider does not require an explicit version. Some providers, such as Microsoft and Anthropic, may require a version string when establishing a connection.

lpszSystemPrompt

A pointer to a null-terminated string that specifies the system prompt used to provide instructions to the model. This parameter may be NULL if no system prompt is required.

nMaxTokens

The maximum number of tokens that may be generated in the response. Specify zero to use the provider or model default.

fTemperature

The temperature value used to control response variability. Lower values produce more predictable responses, while higher values allow more variation. Specify `LLM_TEMPERATURE_DEFAULT` to use the default value.

fTopP

The nucleus sampling value used to limit token selection based on probability. Specify `LLM_TOP_P_DEFAULT` to use the default behavior. Most applications should leave this value at the default unless they specifically need to tune sampling behavior.

lppSession

A pointer to a variable that receives a pointer to an allocated `LLM_SESSION` structure. On successful return, this parameter will contain a pointer to a structure that can be passed to **LlmConnectEx**. The structure must be released using the **LlmDeleteSession** function when it is no longer needed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmCreateSessionEx** function is a helper function that allocates and initializes an `LLM_SESSION` structure using the specified parameters. This simplifies working with session structures, particularly in languages that do not easily support structure initialization or require explicit memory management. This function is typically used in conjunction with **LlmConnectEx** when an application requires extended session configuration.

This function is intended for applications that want to configure commonly used session values without directly initializing the `LLM_SESSION` structure. Applications that need more control can allocate and initialize the structure themselves, or use the **LlmCreateSession** function and modify the returned structure before calling **LlmConnectEx**.

The structure returned by this function is allocated by the library and must be released using the **LlmDeleteSession** function. Failing to release the structure will result in a memory leak.

Applications using languages that do not support pointer-to-pointer parameters may treat the *lppSession* parameter as an opaque pointer value. In this case, the variable should be defined using an integer type that is large enough to hold a pointer value, such as UINT_PTR or an equivalent type. Applications must ensure that the correct pointer size is used for the target platform.

Once a connection has been established using **LlmConnectEx**, the session structure is no longer required and may be released using **LlmDeleteSession**. The library creates an internal copy of the session configuration, so the original structure does not need to remain allocated for the lifetime of the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnectEx, LlmCreateSession, LlmDeleteSession, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmDeleteSession Function

```
BOOL WINAPI LlmDeleteSession(
   LPLLM_SESSION *lppSession
);
```

The **LlmDeleteSession** function releases an `LLM_SESSION` structure allocated by **LlmCreateSession** or **LlmCreateSessionEx**.

Parameters

lppSession

A pointer to a variable that contains a pointer to an `LLM_SESSION` structure previously allocated by **LlmCreateSession** or **LlmCreateSessionEx**. On successful return, the pointer is set to NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmDeleteSession** function releases the memory allocated for an `LLM_SESSION` structure created by **LlmCreateSession** or **LlmCreateSessionEx**. This function should be used only with session structures allocated by the library.

Do not call this function with an `LLM_SESSION` structure allocated by the application, declared on the stack, or embedded in another structure. Only the exact pointer returned by **LlmCreateSession** or **LlmCreateSessionEx** should be passed to this function. Passing any other pointer may cause the application to fail.

After a session structure has been used with **LlmConnectEx**, it may be deleted immediately. The library makes an internal copy of the session configuration when the client session is created, so the original structure does not need to remain allocated for the lifetime of the client handle.

This function does not disconnect an active client session. To close a connection created by **LlmConnect** or **LlmConnectEx**, call **LlmDisconnect**.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    HCLIENT hClient = INVALID_CLIENT;
    LPLLM_SESSION lpSession = NULL;
    DWORD dwError = NO_ERROR;

    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    // Create a session structure using helper function
    if (!LlmCreateSession(LLM_PROVIDER_OPENAI,
```

```
                        NULL,                       // Default base URL
                        _T("gpt-5.4-nano"),         // Model name
                        _T("%OPENAI_API_KEY%"),     // Provider API key
                        NULL,                       // API version
                        &lpSession))
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to create session, error %lu\n"), dwError);
        LlmUninitialize();
        return 1;
    }

    // Establish a connection using the session structure
    hClient = LlmConnectEx(LLM_PROVIDER_OPENAI,
                           LLM_OPTION_DEFAULT,
                           lpSession);

    if (hClient == INVALID_CLIENT)
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);

        // Clean up session before exit
        LlmDeleteSession(&lpSession);
        LlmUninitialize();
        return 1;
    }

    // The session structure is no longer needed after connecting
    LlmDeleteSession(&lpSession);

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse);
    DWORD dwMessageId = 0;

    if (LlmSendMessage(hClient,
                       0,
                       _T("What is the capital city of Denmark?"),
                       szResponse,
                       &dwResponseLength,
                       &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    LlmDisconnect(hClient);
    LlmUninitialize();

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmCreateSession, LlmCreateSessionEx, LlmConnectEx, LlmDisconnect, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmDisableTrace Function

```
BOOL WINAPI LlmDisableTrace();
```

The **LlmDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, a value of zero is returned.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmDisconnect Function

```
INT WINAPI LlmDisconnect(
   HCLIENT hClient
);
```

The **LlmDisconnect** function closes the connection and releases the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmDisconnect** function closes the active connection and releases all memory and resources associated with the specified client session. After this function returns successfully, the client handle is no longer valid and must not be used with any other function.

This function also releases the conversation history, cached message and response data, provider information and any other state maintained for the client session. To preserve information from the session, retrieve it before calling this function.

Applications should call **LlmDisconnect** for every successful call to **LlmConnect** or **LlmConnectEx**. This function should be called before **LlmUninitialize**.

Avoid calling this function while another thread is using the client handle or while a blocking request is in progress. If a request must be canceled, call **LlmCancelRequest** first and wait for the blocking function to return before disconnecting the client session. The **LlmIsBlocking** function can tell you if the library is in the middle of processing a blocking operation, such as retrieving the response to a message prompt.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmCancelRequest, LlmConnect, LlmConnectEx, LlmInitialize, LlmIsBlocking, LlmIsConnected, LlmUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmEnableTrace Function

```
BOOL WINAPI LlmEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **LlmEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmEnumModels Function

```
LONG WINAPI LlmEnumModels(
   HCLIENT hClient,
   LPLLM_MODEL_INFO lpModelList,
   LONG nMaxModels,
   DWORD dwOptions
);
```

The **LlmEnumModels** function returns a list of available models which match the provided options.

Parameters

hClient

A handle to the client session.

lpModelList

A pointer to an array of LLM_MODEL_INFO structures that will receive information about the models which match the specified options. This parameter may be NULL, in which case the function only returns the number of matching models.

nMaxModels

The maximum number of LLM_MODEL_INFO structures in the array specified by the ***lpModelList*** parameter. If ***lpModelList*** is NULL, this value should be zero.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
LLM_MODEL_OPTION_DEFAULT	Returns the default list of chat models for the connected provider.
LLM_MODEL_OPTION_PREVIEW	Include preview or experimental models.
LLM_MODEL_OPTION_STREAMING	Include models that support streaming responses.
LLM_MODEL_OPTION_STRUCTURED	Include models that support structured output, such as JSON mode or schema-constrained responses.
LLM_MODEL_OPTION_REASONING	Include models that support reasoning-oriented generation.
LLM_MODEL_OPTION_VISION	Include models that support image input.
LLM_MODEL_OPTION_AUDIO	Include models that support audio output, such as text-to-speech.
LLM_MODEL_OPTION_IMAGE	Include models that support image generation.
LLM_MODEL_OPTION_VIDEO	Include models that support video generation.
LLM_MODEL_OPTION_TRANSCRIPTION	Include models that support audio transcription, such as speech-to-text.
LLM_MODEL_OPTION_DEPRECATED	Include deprecated, retired, or legacy models

	that are not normally returned.
LLM_MODEL_OPTION_ALL_MODELS	Include all known model types for the connected provider.

Return Value

If the function succeeds, the return value is the number of models which match the specified options. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmEnumModels** function enumerates the models available for the provider associated with the client session. The client must have been successfully connected before calling this function. The returned list is based on the provider metadata available to the library and, when supported by the provider, information returned by the provider itself.

To determine how many models match the specified options, call this function with the *lpModelList* parameter set to NULL and the *nMaxModels* parameter set to zero. If the function succeeds, the return value is the number of matching models. You can then allocate an array of LLM_MODEL_INFO structures and call the function again to retrieve the model information.

Each LLM_MODEL_INFO structure should have its *dwSize* member initialized to the size of the structure before calling this function. This enables the library to validate the structure version and maintain compatibility with future releases.

The model list may be cached for a limited period of time to avoid unnecessary requests to the provider. Use **LlmGetCacheTime** and **LlmSetCacheTime** to determine or change the cache lifetime for the client session.

The model capabilities reported by this function are intended to help an application select an appropriate model. Providers may change their model availability, capabilities, or status over time, and the provider remains the final authority on whether a specific model can be used for a request.

Example

```
LONG nModels = LlmEnumModels(hClient, NULL, 0, LLM_MODEL_OPTION_DEFAULT);

if (nModels > 0)
{
    LPLLM_MODEL_INFO lpModelList;

    lpModelList = (LPLLM_MODEL_INFO)calloc(nModels, sizeof(LLM_MODEL_INFO));

    if (lpModelList != NULL)
    {
        for (LONG nIndex = 0; nIndex < nModels; nIndex++)
            lpModelList[nIndex].dwSize = sizeof(LLM_MODEL_INFO);

        nModels = LlmEnumModels(hClient, lpModelList, nModels,
LLM_MODEL_OPTION_DEFAULT);

        if (nModels != LLM_ERROR)
        {
            for (LONG nIndex = 0; nIndex < nModels; nIndex++)
                _tprintf(_T("%s\n"), lpModelList[nIndex].szName);
        }
```

```
        free(lpModelList);
    }
}
else if (nModels == LLM_ERROR)
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to enumerate models, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetCacheTime, LlmGetDefaultModelName, LlmGetModelInfo, LlmSetCacheTime, LlmValidateModel, LLM_MODEL_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmEnumProviders Function

```
LONG WINAPI LlmEnumProviders(
   LPLLM_PROVIDER_INFO lpProviderList,
   LONG nMaxProviders,
   DWORD dwReserved
);
```

The **LlmEnumProviders** function returns a list of supported service providers.

Parameters

lpProviderList

A pointer to an array of LLM_PROVIDER_INFO structures that will receive information about the supported providers. This parameter may be NULL, in which case the function only returns the number of providers.

nMaxProviders

The maximum number of LLM_PROVIDER_INFO structures in the array specified by the *lpProviderList* parameter. If *lpProviderList* is NULL, this value should be zero.

dwReserved

Reserved for future use. This parameter must be zero.

Return Value

If the function succeeds, the return value is the number of providers. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmEnumProviders** function returns information about the LLM service providers supported by the library. This includes the provider identifier, provider name, host name, default model name, protocol, port number, and capability flags.

To determine how many providers are available, call this function with the *lpProviderList* parameter set to NULL and the *nMaxProviders* parameter set to zero. If the function succeeds, the return value is the number of providers. You can then allocate an array of LLM_PROVIDER_INFO structures and call the function again to retrieve the provider information.

Each LLM_PROVIDER_INFO structure should have its *dwSize* member initialized to the size of the structure before calling this function. This enables the library to validate the structure version and maintain compatibility with future releases.

The list returned by this function only identifies providers that are known to the library. It does not verify that the provider is currently reachable, that an account is active, or that a valid API key has been configured.

Example

```
LONG nProviders = LlmEnumProviders(NULL, 0, 0);

if (nProviders > 0)
{
    LPLLM_PROVIDER_INFO lpProviderList;

    lpProviderList = (LPLLM_PROVIDER_INFO)calloc(nProviders,
sizeof(LLM_PROVIDER_INFO));
```

```
    if (lpProviderList != NULL)
    {
        for (LONG nIndex = 0; nIndex < nProviders; nIndex++)
            lpProviderList[nIndex].dwSize = sizeof(LLM_PROVIDER_INFO);

        nProviders = LlmEnumProviders(lpProviderList, nProviders, 0);

        if (nProviders != LLM_ERROR)
        {
            for (LONG nIndex = 0; nIndex < nProviders; nIndex++)
                _tprintf(_T("%s\n"), lpProviderList[nIndex].szDisplayName);
        }

        free(lpProviderList);
    }
}
else if (nProviders == LLM_ERROR)
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to enumerate providers, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetDefaultModelName, LlmGetProviderFlags, LlmGetProviderInfo, LlmGetProviderName, LLM_PROVIDER_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmEstimateTokenCount Function

```
LONG WINAPI LlmEstimateTokenCount(
   DWORD dwProviderId
   LPCTSTR lpszMessageText,
   INT cchMessageText
);
```

The **LlmEstimateTokenCount** function estimates the number of tokens in a string.

Parameters

dwProviderId

The identifier of the provider used to estimate the token count. This value determines the tokenization rules that are applied, as different providers may use different tokenization algorithms.

lpszMessageText

A pointer to a null-terminated string containing the text to be analyzed. This parameter cannot be NULL.

cchMessageText

The number of characters in the string specified by the ***lpszMessageText*** parameter. If this value is -1, the string is assumed to be null-terminated.

Return Value

If the function succeeds, the return value is the estimated number of tokens in the specified string. If there are no text characters, or the string only contains whitespace, the function will return zero. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmEstimateTokenCount** function provides an approximate count of the number of tokens that would be required to represent the specified text for a given provider. This can be useful when estimating prompt size and ensuring that requests remain within model limits.

The ***lpszMessageText*** parameter must contain valid Unicode text and cannot contain arbitrary binary data or embedded null characters. If malformed Unicode is detected, the function will fail and return LLM_ERROR, with the last error code set to ST_ERROR_INVALID_UNICODE. If the string consists only of whitespace characters, including non-breaking spaces, the function will return zero.

The ANSI version of this function expects the input string to contain either ASCII or UTF-8 encoded text. Strings encoded using a localized ANSI code page are not supported and may fail validation if they contain non-ASCII characters. To ensure internationalized text is processed correctly, it is recommended that you use the Unicode version of this function whenever possible.

A token is a unit of text used internally by large language models. Tokens are not the same as characters or words. In general, a token may represent a single word, part of a word, or a short sequence of characters. For example, common words may be represented as a single token, while less common or longer words may be split into multiple tokens. Punctuation and whitespace may also be represented as separate tokens.

As a general rule of thumb, one token corresponds to approximately three to four characters of English text. This can vary significantly depending on the content, language, and tokenization rules used by the provider.

Most providers distinguish between input tokens (the text sent to the model) and output tokens (the text generated by the model in response). Usage and billing are typically based on the total number of tokens processed, including both the prompt and the generated response. Monitoring token usage can help applications control costs and ensure that requests remain within the limits imposed by the selected model.

The tokenization rules used by each provider can vary, and the exact number of tokens used in a request is determined by the provider at runtime. The value returned by this function is an estimate only and should not be relied on as an exact measurement.

This function does not require an active client session and can be used independently to estimate token usage before establishing a connection or sending a request.

To determine how many input and output tokens have been used in a session, use the **LlmGetTokenUsage** function.

Example

```
LPCTSTR lpszMessageText = _T("Explain how a TCP connection is established.");

LONG nTokens = LlmEstimateTokenCount(LLM_PROVIDER_OPENAI, lpszMessageText, -1);

if (nTokens != LLM_ERROR)
{
    _tprintf(_T("Estimated token count: %d\n"), nTokens);
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to estimate token count, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetTokenUsage, LlmNormalizeText, LlmSendMessage, LlmSendMessageEx, LlmValidateText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmExportHistory Function

```
BOOL WINAPI LlmExportHistory(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LONG nMessageCount,
   LPLONG lpnExported
);
```

The **LlmExportHistory** function exports the conversation history to a JSON-formatted file.

Parameters

hClient

A handle to the client session.

lpszFileName

A pointer to a null-terminated string which specifies the name of the file to be created. This parameter cannot be NULL or a zero-length string.

nMessageCount

The number of the most recent messages and responses which should be exported. If this value is zero, all messages in the current conversation are exported.

lpnExported

A pointer to an integer which will receive the number of messages which were exported when the function returns. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmExportHistory** function exports the current conversation history associated with the client session to a JSON-formatted file. The exported history includes both client messages and model responses stored in the current session history.

This function only exports conversation history currently stored in memory for the client session. It cannot be used if the conversation history has been disabled by setting the history size to zero.

If the *lpszFileName* parameter specifies a directory or device name, the function will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters, they will be removed. If the file already exists, the current process must have permission to overwrite the file. If the file has the read-only attribute set, the function will fail and the last error code will be set to `ST_ERROR_ACCESS_DENIED`.

If the *nMessageCount* parameter is non-zero, the function exports only the most recent messages in the conversation history. Older messages are skipped. If the specified count exceeds the number of available messages, all available messages are exported.

The exported file uses UTF-8 encoded JSON text and is intended to be human-readable. Applications should not depend on the exact formatting or internal structure of the file because future versions of the library may include additional metadata or fields. To help prevent data loss, the function writes the exported history to a temporary file and only replaces the destination file after the export operation

has completed successfully. If the export fails, any existing file specified by the *lpszFileName* parameter is left unchanged.

The exported file is intended to provide a simple mechanism for persisting conversation state between application sessions. A previously exported history file can later be restored using the **LlmImportHistory** function.

Important: The conversation history may contain sensitive or private information. Applications should take reasonable precautions to protect exported history files, such as limiting access permissions or encrypting the exported file. Applications which allow users to export conversation history should consider informing users that the exported file may contain private information.

Example

```
LONG nExported = 0;

// Export the complete conversation history to a JSON file
if (LlmExportHistory(hClient,
                     _T("%USERPROFILE%\\Documents\\conversation.json"),
                     0,
                     &nExported))
{
    _tprintf(_T("Exported %ld messages\n"), nExported);
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to export conversation history, error 0x%08lX\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetMessageCount, LlmImportHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmFindMessage Function

```
DWORD WINAPI LlmFindMessage(
   HCLIENT hClient,
   LPCTSTR lpszPattern,
   DWORD dwStartMessageId,
   DWORD dwOptions
);
```

The **LlmFindMessage** function searches the client conversation history for matching text.

Parameters

hClient

A handle to the client session.

lpszPattern

A null-terminated string which specifies the text that should be searched for. This may be a literal string or it may specify a simplified regular expression. The search options can be used to specify if the search is case-sensitive. This parameter cannot be NULL or a string which only contains whitespace.

dwStartMessageId

An unsigned integer which specifies the message ID that the search should begin with. If this parameter value is zero, the search begins with the oldest message in the client history.

dwOptions

An unsigned integer that specifies one or more search options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
LLM_SEARCH_DEFAULT	Search the entire message history for matching text.
LLM_SEARCH_PROMPT	Search the history of message prompts for matching text.
LLM_SEARCH_CASE_SENSITIVE	The pattern match will be case-sensitive. If this option is not specified, matches are not case-sensitive.
LLM_SEARCH_LITERAL_MATCH	Perform a literal string search instead of using a regular expression.

Return Value

If a matching message is found, the function returns a non-zero message ID. If no message prompts and/or responses match the search pattern, the function returns zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmFindMessage** function searches the message history for text that matches a specific pattern. The search can be limited to prompts, responses, or both, depending on the options specified by the *dwOptions* parameter. The pattern you are matching against must be a null-terminated string and it cannot consist of only whitespace characters (spaces, tabs, carriage returns and linefeeds).

If the *dwStartMessageId* parameter is non-zero, the function will begin searching with the first message whose identifier is equal to or greater than the specified value. Message identifiers are unique within a client session and are assigned incrementally, so newer messages will always have

higher identifier values than older messages.

If the ***lpszPattern*** parameter does not contain any regular expression constructs (such as **^**, **$**, **[ ]**, or quantifiers like ***** and **+**), the search is performed using a simple substring comparison. Regular expression matching is automatically enabled when the pattern contains recognized regular expression constructs, unless the LLM_SEARCH_LITERAL_MATCH option is specified.

If a regular expression pattern is specified, the function compiles the pattern and searches the message content for a match. The regular expression matching uses a simplified version which only supports anchors, wildcards, match classes (enclosed in brackets) and a limited number of escape sequences such as **\d** to match any digit or **\s** to match any space character. It does not support grouping or replacement.

Supported regular expression constructs include:

- The caret **^** and dollar symbol **$** for the beginning and end of the content.
- The period **.** to match any character.
- Character classes such as **[abc]** which would match any of the characters enclosed in brackets.
- Negated classes such as **[^abc]** which would match if none of those characters are found.
- Escaped matches such as **\d** for digits, **\w** for word characters and **\s** for whitespace characters
- The **+** quantifier can be used to match one or more occurrences.
- The ***** quantifier will match zero or more occurrences.

This function uses a minimal regular expression parser and does not include support for complex constructs such as grouping, backreferences or alternation using the pipe **|** character. If you want to search for text which may contain characters typically used in regular expressions such as **[** or **$**, you can either escape them using a backslash, or you can use the LLM_SEARCH_LITERAL_MATCH option which forces the function to only perform a substring search.

Applications searching for arbitrary user-entered text should usually specify the LLM_SEARCH_LITERAL_MATCH option to avoid unintended interpretation of regular expression characters. This will avoid situations where a user may unexpectedly enter search text which appears to be a regular expression pattern.

Regular expression matches are performed against the entire prompt or response text and are not implicitly anchored. It is also important to keep in mind that matches can span multiple lines (they do not end at line breaks) and will attempt to match as much text as possible (this is sometimes referred to as a "greedy match"). This can yield unexpected results if you use wildcards in the pattern which are overly broad, matching large sections of the prompt or response text. For example, a pattern like ".*abc" will match everything until the last instance of "abc" is found.

If you are calling this function in C, C++ or another C-like language and want to use a backslash in your regular expression, remember you will need to escape the backslash (in other words, use double backslashes). For example, if you wanted to use the regular expression \d+\s+ in your pattern, the C string literal would look like "\\d+\\s+", otherwise the compiler will probably warn you about unrecognized escape sequences.

If neither the LLM_SEARCH_PROMPT nor LLM_SEARCH_RESPONSE options are explicitly specified, the search will match against both message prompts and responses.

The **LlmGetMessageById** function can be used to obtain the complete contents of the prompt and/or response for a matching message.

Example

```
LPCTSTR lpszPattern = _T("exception.*handler");
DWORD dwOptions = LLM_SEARCH_DEFAULT;
DWORD dwMessageId = 0;

// Find the first message that contains the specified text
dwMessageId = LlmFindMessage(hClient, lpszPattern, dwMessageId, dwOptions))

if (dwMessageId != 0)
{
    _tprintf(_T("Found match in message ID %lu\n"), dwMessageId);

    // Continue searching for additional matches
    do
    {
        dwMessageId = LlmFindMessage(hClient, lpszPattern, dwMessageId + 1,
dwOptions);

        if (dwMessageId != 0)
            _tprintf(_T("Found another match in message ID %lu\n"), dwMessageId);
    }
    while (dwMessageId != 0);
}
else
{
    _tprintf(_T("No matching messages were found.\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetFirstMessage, LlmGetHistorySize, LlmGetMessageById, LlmGetMessageCount, LlmGetNextMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmFormatTimestamp Function

```
BOOL INT LlmFormatTimestamp(
   TIME_T tTimestamp,
   LPTSTR lpszDateTime,
   INT nMaxLength,
   BOOL bLocalTime
);
```

The **LlmFormatTimestamp** function converts a timestamp value to an ISO 8601 formatted date and time string.

Parameters

tTimestamp

A 64-bit timestamp value associated with a message or response. This value represents the number of seconds that have elapsed since January 1, 1970 UTC.

lpszDateTime

A pointer to a buffer that receives the formatted date and time string. This parameter cannot be NULL and must be large enough to store the entire string, including the terminating null character.

nMaxLength

The size of the buffer specified by the lpszDateTime parameter, in characters. This value must be large enough to store the formatted string. A buffer of at least 32 characters is recommended.

bLocalTime

If this parameter is non-zero, the timestamp is converted to the local system time before formatting. If this parameter is zero, the timestamp is formatted as Coordinated Universal Time (UTC).

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmFormatTimestamp** function converts a timestamp value returned by the library into a human-readable date and time string using the ISO 8601 format. This format is consistent, unambiguous, and suitable for logging, display, or storage.

The *tTimestamp* parameter is a standard 64-bit Unix time value and can be used interchangeably with functions in the C standard library such as **`gmtime`** or **`localtime`**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps are returned by functions such as **LlmSendMessage** and can be converted into a local date and time using the **LlmConvertTimestamp** function.

The formatted string is returned in the form "YYYY-MM-DDThh:mm:ssZ" when UTC is specified. When local time is specified, the string is returned in the form "YYYY-MM-DDThh:mm:ss+hh:mm" or "YYYY-MM-DDThh:mm:ss-hh:mm", depending on the current time zone offset.

When *bLocalTime* is non-zero, the formatted string includes the local time zone offset. This ensures that the resulting value is not ambiguous and preserves the ISO 8601 format.

Applications that require locale-specific formatting or alternative date and time representations should convert the timestamp using **LlmConvertTimestamp** and apply additional formatting as needed. If you need to display the date and time using conventions for the current locale, use functions such as

GetDateFormatEx and **GetTimeFormatEx** in the Windows API.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };

if (LlmGetFirstMessage(hClient, 0, &clientMessage, &serverResponse))
{
    do
    {
        TCHAR szDateTime[64] = {0};

        LlmFormatTimestamp(clientMessage.tTimestamp, szDateTime,
_countof(szDateTime), FALSE);

        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Timestamp:       %s\n"), szDateTime);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        LlmFormatTimestamp(serverResponse.tTimestamp, szDateTime,
_countof(szDateTime), FALSE);

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Timestamp:       %s\n"), szDateTime);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (LlmGetNextMessage(hClient, &clientMessage, &serverResponse));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConvertTimestamp, LlmGetFirstMessage, LlmGetNextMessage, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetCacheTime Function

```
INT WINAPI LlmGetCacheTime(
   HCLIENT hClient
);
```

The **LlmGetCacheTime** function returns the current cache lifetime, in seconds, for the specified client session.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the current cache time in seconds. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetCacheTime** function retrieves the amount of time, in seconds, that provider and model information is cached by the client session. Caching this information reduces the number of requests made to the provider and can improve performance when repeatedly querying for models or provider metadata.

By default, the cache lifetime for a client session is one hour (3600 seconds). The cache time can be adjusted using the **LlmSetCacheTime** function. Shorter cache times will refresh provider and model information more frequently, while longer cache times can reduce network activity.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetIdleTime, LlmGetProviderInfo, LlmGetTimeout, LlmSetCacheTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetClientOptions Function

```
BOOL WINAPI LlmGetClientOptions(
    HCLIENT hClient,
    LPDWORD lpdwOptions
);
```

The **LlmGetClientOptions** function returns the current client option flags.

Parameters

hClient

A handle to the client session.

lpdwOptions

A pointer to an unsigned integer that will be set to the current client options when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetClientOptions** function returns the current option flags for the client session. The variable specified by the *lpdwOptions* parameter will be initialized to LLM_OPTION_NONE and then updated with the client options before the function returns. The value may contain one or more of the following flags combined with a bitwise OR operation.

Value	Description
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be used when a provider does not require a User-Agent

	header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.

This function can be used in conjunction with **LlmSetClientOptions** to query and modify client behavior at runtime.

Example

```
DWORD dwProviderId = LLM_PROVIDER_NONE;
TCHAR szBaseUrl[MAX_PATH] = {0};

// Determine the provider ID for the current session and get
// the base URL used for API requests
dwProviderId = LlmGetClientProvider(hClient, szBaseUrl, _countof(szBaseUrl));

if (dwProviderId == LLM_PROVIDER_NONE)
{
    _tprintf(_T("Unable to get provider ID for this session\n"));
    return;
}

// Get the current client options
DWORD dwOptions = LLM_OPTION_NONE;

if (!LlmGetClientOptions(hClient, &dwOptions))
{
    _tprintf(_T("Unable to get client options for this session\n"));
    return;
}

// Get information about the current provider
LLM_PROVIDER_INFO providerInfo = { sizeof(providerInfo), 0 };

if (!LlmGetProviderInfo(dwProviderId, &providerInfo))
{
    _tprintf(_T("Unable to get provider information\n"));
    return;
}

_tprintf(_T("Connected to %s using %s, the connection is %s\n"),
         providerInfo.szDisplayName,
         szBaseUrl,
         ((dwOptions & LLM_OPTION_SECURE) ? _T("secure") : _T("not secure")));
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetClientProvider, LlmGetProviderInfo, LlmSetClientOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetClientProvider Function

```
DWORD WINAPI LlmGetClientProvider(
    HCLIENT hClient,
    LPCTSTR lpszBaseUrl,
    INT nMaxLength
);
```

The **LlmGetClientProvider** function returns the provider associated with the specified client session and, optionally, the base URL used for requests.

Parameters

hClient

A handle to the client session.

lpszBaseUrl

A pointer to a buffer that receives the base URL used by the client session when making requests to the provider. This parameter may be NULL if the base URL is not required.

nMaxLength

The size of the buffer specified by the ***lpszBaseUrl*** parameter, in characters. If ***lpszBaseUrl*** is NULL, this value should be zero. If a buffer is provided, it must be large enough to store the complete URL, including the terminating null character.

Return Value

If the function succeeds, the return value is the provider ID. If the function fails, the return value is LLM_PROVIDER_UNKNOWN. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetClientProvider** function can be used to determine which provider is currently associated with a client session. This is useful when an application supports multiple providers and needs to adjust its behavior based on the active provider.

If the ***lpszBaseUrl*** parameter is specified, the function also returns the base URL used by the client when communicating with the provider. This may be a default URL defined by the library or a custom URL specified when the session was created or connected.

The base URL is returned as a null-terminated string. If the buffer is not large enough to receive the entire string, the function fails and sets the error code to ST_ERROR_BUFFER_TOO_SMALL. We recommend a buffer size of at least 256 characters when retrieving the base URL.

To determine the default endpoint URL used by a specific service provider, use the **LlmGetProviderUrl** function.

Example

```
TCHAR szBaseUrl[256] = {0};

DWORD dwProviderId = LlmGetClientProvider(hClient, szBaseUrl, _countof(szBaseUrl));

if (dwProviderId != LLM_PROVIDER_UNKNOWN)
{
    _tprintf(_T("Provider ID: %lu\n"), dwProviderId);
    _tprintf(_T("Base URL: %s\n"), szBaseUrl);
}
```

```
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to get provider information, error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetProviderFlags, LlmGetProviderId, LlmGetProviderInfo, LlmGetProviderName, LlmGetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetDefaultModelFlags Function

```
BOOL WINAPI LlmGetDefaultModelFlags(
    DWORD dwProviderId
    LPCTSTR lpszModelName,
    LPDWORD lpdwFlags
);
```

The **LlmGetDefaultModelFlags** function returns the default capability flags for the specified model.

Parameters

dwProviderId

The identifier of the provider associated with the model.

lpszModelName

A null-terminated string which contains the name of the new model. If this parameter is NULL or a zero-length string, the default model for the specified provider will be used.

lpdwFlags

A pointer to an unsigned integer which may have one or more bit flags set when the function returns. If there are no capabilities defined for the specified model, this value will be set to `LLM_MODEL_FLAG_NONE`. This parameter cannot be NULL and the value does not need to be initialized by the caller.

Constant	Description
LLM_MODEL_FLAG_NONE	No model capabilities defined
LLM_MODEL_FLAG_CHAT	Supports conversational text generation
LLM_MODEL_FLAG_SYSTEM_ROLE	Accepts a system role message in the input
LLM_MODEL_FLAG_STREAMING	Supports streaming responses
LLM_MODEL_FLAG_STRUCTURED	Supports structured output (e.g., JSON or schema)
LLM_MODEL_FLAG_VISION	Supports image inputs (multimodal)
LLM_MODEL_FLAG_TOOLS	Supports tool or function calling
LLM_MODEL_FLAG_REASONING	Supports reasoning-oriented generation
LLM_MODEL_FLAG_RESPONSES	Supports the Responses API (unified endpoint)
LLM_MODEL_FLAG_FUNCTIONS	Supports OpenAI-style function calling with schemas
LLM_MODEL_FLAG_AUDIO	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	Supports image generation
LLM_MODEL_FLAG_VIDEO	Supports video generation
LLM_MODEL_FLAG_TRANSCRIPTION	Supports speech-to-text transcription

LLM_MODEL_FLAG_INSTRUCTIONS	Supports a dedicated instruction field outside message lists
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	Uses `max_completion_tokens` instead of `max_tokens` in requests
LLM_MODEL_FLAG_INFERRED	Capabilities are inferred rather than explicitly defined

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetDefaultModelFlags** function returns the default capability flags for a model associated with the specified provider. If the model name is recognized, the library will return the inferred capabilities associated with that model. If the model name is not recognized, the *dwFlags* parameter will return `LLM_MODEL_FLAG_NONE`.

This function does not require the application to be connected to a service provider. These capabilities are inferred from an internal table of recognized models and may not always reflect the current behavior of a provider, particularly when providers update, rename or replace models. The **LlmGetModelFlags** function can be used to determine the capabilities of the model that was selected when a connection was established with a provider.

Because the capability flags are derived from the model name rather than explicitly provided by the service provider, the `LLM_MODEL_FLAG_INFERRED` flag will be set for recognized models.

To obtain the name of the default model the library uses for a specific provider, call the **LlmGetDefaultModelName** function.

Example

```
DWORD dwProviderId = LLM_PROVIDER_OPENAI;
TCHAR szModelName[LLM_MODEL_NAMELEN] = _T("");

if (LlmGetDefaultModelName(dwProviderId, szModelName, _countof(szModelName)) > 0)
{
    DWORD dwFlags = LLM_MODEL_FLAG_NONE;

    if (LlmGetDefaultModelFlags(dwProviderId, szModelName, &dwFlags))
    {
        if (dwFlags & LLM_MODEL_FLAG_REASONING)
            _tprintf(_T("The %s model has reasoning capability\n"), szModelName);
    }
    else
    {
        DWORD dwError = LlmGetLastError();
       _tprintf(_T("Unable to retrieve default model flags, error 0x%08lx\n"),
dwError);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEnumModels, LlmGetDefaultModelName, LlmGetModelInfo, LlmValidateModel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetDefaultModelName Function

```
INT WINAPI LlmGetDefaultModelName(
    DWORD dwProviderId
    LPTSTR lpszModelName,
    INT nMaxLength
);
```

The **LlmGetDefaultModelName** function returns the name of the default model for a service provider.

Parameters

dwProviderId

The identifier of the provider whose default model name will be returned.

lpszModelName

A pointer to a buffer that receives the default model name. This parameter may be NULL, in which case the function returns the length of the model name without copying it.

nMaxLength

The size of the buffer specified by the ***lpszModelName*** parameter, in characters. If ***lpszModelName*** is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetDefaultModelName** function returns the default model name associated with the specified provider. This is the model that will be used if a session is created or connected without explicitly specifying a model name. In most cases, the default model is selected to provide a balance of performance and cost, typically favoring smaller, faster models that are suitable for general use. For more information about providers and their default models, refer to the Large Language Model Providers section.

Applications can call this function to determine a suitable default model for a provider, or to present a default selection to the user. The default model may change over time as providers update or replace their models.

If the ***lpszModelName*** parameter is NULL, the function returns the length of the model name, in characters, not including the terminating null character. This allows an application to determine the required buffer size before retrieving the string.

The model name is returned as a null-terminated string. If a buffer is provided but is not large enough to receive the entire string, the function fails and sets the error code to ST_ERROR_BUFFER_TOO_SMALL. A buffer of 128 characters is typically sufficient for most model names.

Example

```
INT cchModelName = LlmGetDefaultModelName(LLM_PROVIDER_OPENAI, NULL, 0);

if (cchModelName > 0)
```

```
{
    LPTSTR lpszModelName;
    lpszModelName = (LPTSTR)malloc((cchModelName + 1) * sizeof(TCHAR));

    if (lpszModelName != NULL)
    {
        if (LlmGetDefaultModelName(LLM_PROVIDER_OPENAI, lpszModelName, cchModelName
+ 1) > 0)
        {
             _tprintf(_T("The default model is %s\n"), lpszModelName);
        }

        free(lpszModelName);
    }
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to retrieve default model name, error 0x%08lx\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEnumModels, LlmGetDefaultModelFlags, LlmGetModelInfo, LlmValidateModel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetErrorString Function

```
INT WINAPI LlmGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **LlmGetErrorString** function returns the description of a specific error code.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Remarks

This function is used in conjunction with the **LlmGetLastError** function for use with warning dialogs or as diagnostic messages. Applications should not rely on the returned description text for program logic. Use the numeric error code for comparisons and handling specific conditions.

The error descriptions returned by this function are provided in English only. Applications that require localized error messages should provide their own translations based on the returned error code.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetLastError, LlmSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetFirstMessage Function

```
BOOL WINAPI LlmGetFirstMessage(
   HCLIENT hClient,
   DWORD dwReserved,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **LlmGetFirstMessage** function returns the first available message in the client history.

Parameters

hClient

A handle to the client session.

dwReserved

An integer value reserved for future use. This parameter should always be zero.

lpMessage

A pointer to an LLM_MESSAGE structure that receives information about the client request. This parameter may be NULL, in which case no message information is returned.

lpResponse

A pointer to an LLM_RESPONSE structure that receives information about the server response. This parameter may be NULL, in which case no response information is returned.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**. If there are no messages in the client history, the error code will be set to `ST_ERROR_END_OF_MESSAGES`.

Remarks

The **LlmGetFirstMessage** function retrieves the first message and associated response from the client session history. This function is typically used in conjunction with the **LlmGetNextMessage** function to enumerate all messages in the session.

If the *lpMessage* and/or *lpResponse* parameters are specified, their respective structures must be initialized with the *dwSize* member set to the size of the structure before calling this function. This enables the library to validate the structure version and maintain compatibility with future releases.

Messages are returned in chronological order, beginning with the oldest message in the client history. New messages are always appended to the end of the history, and if the history is trimmed, older messages are removed first. The order of messages is stable and reflects the sequence in which they were submitted.

If there are no messages in the client history, the function fails and sets the error code to ST_ERROR_END_OF_MESSAGES. Applications can use this condition to determine that no messages are available for enumeration.

Either the *lpMessage* or *lpResponse* parameter may be NULL if only one type of information is required. At least one of these parameters must be specified. If both parameters are NULL, the function fails and sets the error code to ST_ERROR_INVALID_PARAMETER.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };

if (LlmGetFirstMessage(hClient, 0, &clientMessage, &serverResponse))
{
    do
    {
        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (LlmGetNextMessage(hClient, &clientMessage, &serverResponse));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetNextMessage, LlmGetHistorySize, LlmGetMessageById

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetHistorySize Function

```
LONG WINAPI LlmGetHistorySize(
    HCLIENT hClient
);
```

The **LlmGetHistorySize** function returns the maximum number of message exchanges that can be stored in the conversation history.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the maximum size of the conversation history for the client session. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetHistorySize** function returns the maximum number of messages which can be stored in the client conversation history. Each message represents a single request submitted to the model and its associated response.

The conversation history is maintained on a per-session basis and reflects the sequence of interactions performed using the client handle. Messages are stored in chronological order, with new messages appended to the end of the history. To retrieve the contents of a specific message and response from the history, use either the **LlmGetMessage** or **LlmGetMessageById** functions.

It is possible to disable the conversation history by calling the **LlmSetHistorySize** function and specifying a maximum history size of zero. In this case, this function will always return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetFirstMessage, LlmGetMessage, LlmGetMessageById, LlmGetNextMessage, LlmSetHistorySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetIdleTime Function

```
INT WINAPI LlmGetIdleTime(
   HCLIENT hClient
);
```

The **LlmGetIdleTime** function returns the current idle time for the specified client session.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the current idle timeout period in seconds. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetIdleTime** function returns the current idle timeout period, in seconds, for the specified client session. This value determines how long a connection can remain idle between requests before the library automatically closes and re-establishes the underlying HTTP connection.

Idle time is used to balance performance and reliability. Reusing an existing connection can reduce overhead and improve performance, particularly when multiple requests are made in quick succession. However, connections that remain idle for extended periods may be closed by the server or by intermediate network infrastructure. When the idle time is exceeded, the library will proactively establish a new connection before sending the next request.

This value is distinct from the timeout period and cache time. The timeout period specifies how long the client will wait for a response to a request before failing, while the cache time controls how long certain responses or metadata may be reused without making a new request. The idle time only affects how long an existing connection is retained between requests.

The idle time can be modified using the **LlmSetIdleTime** function. If a value of zero is specified, a new connection is established for each request. In most cases, the default idle time provides a good balance between performance and compatibility.

It is important to note that this setting only affects client-side behavior. Increasing the idle time does not guarantee that a server will keep the connection open for that duration. Many servers will close idle connections after a shorter period of time, in which case the library will automatically reconnect as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetCacheTime, LlmGetTimeout, LlmSetIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetLastError Function

```
DWORD WINAPI LlmGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **LlmGetLastError** function immediately when a function's return value indicates that an error has occurred. Functions set this value by calling **LlmSetLastError**. The Return Value section of each reference page notes the conditions under which a function sets the last error code.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or LLM_ERROR. Those functions which call **LlmSetLastError** when they succeed are noted on the function reference page.

This function should only be called when a preceding function indicates failure. Calling this function after a successful call may return a value unrelated to the most recent operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetErrorString, LlmSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetMemoryUsage Function

```
BOOL WINAPI LlmGetMemoryUsage(
   HCLIENT hClient,
   PLARGE_INTEGER lpMemUsage
);
```

The **LlmGetMemoryUsage** function returns the number of bytes of memory allocated for the client session.

Parameters

hClient

A handle to the client session.

lpMemUsage

A pointer to a `LARGE_INTEGER` variable which will contain the number of bytes allocated for the client session when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetMemoryUsage** function returns an approximate amount of memory currently allocated for the specified client session. This includes the client session structure, dynamically allocated session data, cached model information, and message history retained by the session.

This function is primarily intended for diagnostics, monitoring, and testing. Applications that maintain long-running sessions or retain large message histories can use this function to monitor how much memory is being used by a client session.

The value returned by this function is rounded up to the nearest system page size and should be treated as an estimate of memory usage rather than an exact byte-for-byte allocation total.

The *lpMemUsage* parameter points to a LARGE_INTEGER structure. The memory usage value is returned in the QuadPart member. In languages that support 64-bit integer values directly, it is safe to pass a pointer to a compatible 64-bit integer variable.

The memory usage reported by this function is limited to memory allocated for the specified client session. It does not include memory used internally by the operating system, networking libraries, provider services, or other process-wide resources.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmClearHistory, LlmGetHistorySize, LlmSetHistorySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetMessage Function

```
BOOL WINAPI LlmGetMessage(
   HCLIENT hClient,
   LONG nMessageIndex,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **LlmGetMessage** function retrieves a message turn in the client history using an index value.

Parameters

hClient

A handle to the client session.

nMessageIndex

An integer value which identifies the message using a zero-based index value. This value cannot be less than zero or greater than the number of messages in the conversation history.

lpMessage

A pointer to an LLM_MESSAGE structure that receives information about the client request. This parameter may be NULL, in which case no message information is returned.

lpResponse

A pointer to an LLM_RESPONSE structure that receives information about the server response. This parameter may be NULL, in which case no response information is returned.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetMessage** function retrieves a message and its corresponding response from the client history using a zero-based index value. The index value of zero refers to the oldest message currently stored in the history.

The message history is managed as a first-in, first-out (FIFO) queue. When the number of messages exceeds the configured history size, older messages are automatically removed. As a result, the index value for a given message may change over time. Applications that need to reliably reference a specific message should use the **LlmGetMessageById** function instead.

If message history has been disabled, the library does not retain previous messages. However, the most recent message and response are still available and may be retrieved using an index value of zero. This also means the data returned in the LLM_MESSAGE and LLM_RESPONSE structures will only remain valid until the next call to **LlmSendMessage**, **LlmSendMessageEx**, or until the session is reset or disconnected. Applications that need to preserve this information should make a copy of the data before issuing another request.

If the *nMessageIndex* parameter is less than zero or greater than or equal to the number of messages currently stored in the history, the function will fail and set the error code to ST_ERROR_INVALID_PARAMETER.

If the *lpMessage* and/or *lpResponse* parameters are specified, their respective structures must be initialized with the *dwSize* member set to the size of the structure before calling this function. This

enables the library to validate the structure version and maintain compatibility with future releases.

Either the *lpMessage* or *lpResponse* parameter may be NULL if only one type of information is required. At least one of these parameters must be specified. If both parameters are NULL, the function fails and sets the error code to ST_ERROR_INVALID_PARAMETER.

Example

```
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE) };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE) };
LONG nMessageIndex = 0;

// If conversation history is enabled, retrieve the most recent message.
// If history is disabled, an index of 0 refers to the most recent
// message and response from the previous request.

LONG nCount = LlmGetMessageCount(hClient);

if (LlmGetHistorySize(hClient) > 0 && nCount > 0)
    nMessageIndex = nCount - 1;

if (LlmGetMessage(hClient, nMessageIndex, &clientMessage, &serverResponse))
{
    _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
    _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
    _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
    _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

    _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
    _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
    _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
    _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
    _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
    _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
    _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
    _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to retrieve message at index %ld (error 0x%08lX)\n"),
nMessageIndex, dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetFirstMessage, LlmGetMessageById, LlmGetNextMessage, LlmSendMessage, LlmSendMessageEx, LlmSetHistorySize, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetMessageById Function

```
BOOL WINAPI LlmGetMessageById(
   HCLIENT hClient,
   DWORD dwMessageId,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **LlmGetMessageById** function retrieves a message turn in the client history based on a numeric identifier.

Parameters

hClient

A handle to the client session.

dwMessageId

A numeric value which identifies a specific message and response.

lpMessage

A pointer to an LLM_MESSAGE structure that receives information about the client request. This parameter may be NULL, in which case no message information is returned.

lpResponse

A pointer to an LLM_RESPONSE structure that receives information about the server response. This parameter may be NULL, in which case no response information is returned.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetMessageById** function retrieves a specific message and its associated response from the client session history. This function can be used when an application has stored a message identifier or has obtained one using functions such as **LlmFindMessage**. Message identifiers are returned when sending requests using **LlmSendMessage** or **LlmSendMessageEx**.

If message history has been disabled, the library does not retain previous messages. However, the most recent message and response are still available and may be retrieved specifying its message identifier. This also means the data returned in the LLM_MESSAGE and LLM_RESPONSE structures will only be valid until the next call to **LlmSendMessage**, **LlmSendMessageEx**, or until the session is reset or disconnected. Applications that need to preserve this information should make a copy of the data before issuing another request.

If the *lpMessage* and/or *lpResponse* parameters are specified, their respective structures must be initialized with the *dwSize* member set to the size of the structure before calling this function. This enables the library to validate the structure version and maintain compatibility with future releases.

Either the *lpMessage* or *lpResponse* parameter may be NULL if only one type of information is required. At least one of these parameters must be specified. If both parameters are NULL, the function fails and sets the error code to ST_ERROR_INVALID_PARAMETER.

Message identifiers are session-specific and are guaranteed to be unique only within the client session in which they were created. Applications must not rely on the internal assignment strategy used to

generate these identifiers. The only guaranteed behavior is that message identifiers are assigned values which ensure newer messages always receive larger values than older messages.

If the specified message identifier does not match a message currently stored in the client history, the function fails. This can occur if the message has been removed because the history was cleared or trimmed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmFindMessage, LlmGetFirstMessage, LlmGetMessage, LlmGetNextMessage, LlmSendMessage, LlmSendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetMessageCount Function

```
LONG WINAPI LlmGetMessageCount(
    HCLIENT hClient
);
```

The **LlmGetMessageCount** function returns the current number of messages in the client history.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the number of messages in the client history. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetMessageCount** function returns the number of messages currently stored in the client session history. Each message represents a single request submitted to the model and its associated response.

The conversation history is maintained on a per-session basis and reflects the sequence of interactions performed using the client handle. Messages are stored in chronological order, with new messages appended to the end of the history. To retrieve the contents of a specific message and response from the history, use either the **LlmGetMessage** or **LlmGetMessageById** functions.

It is possible to disable the conversation history by calling the **LlmSetHistorySize** function and specifying a maximum history size of zero. In this case, this function will always return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmClearHistory, LlmGetMessage, LlmGetMessageById, LlmSetHistorySize, LlmTrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetModelContextSize Function

```
LONG WINAPI LlmGetModelContextSize(
    HCLIENT hClient,
    LPCTSTR lpszModelName
);
```

The **LlmGetModelContextSize** function returns the context size for the specified model.

Parameters

hClient

A handle to the client session. The provider associated with this session is used to determine the context size for the specified model.

lpszModelName

A pointer to a null-terminated string that specifies the name of the model. This parameter may be NULL or an empty string, in which case the context size is returned for the model selected for the current client session. If a model name is specified, it must be valid for the provider associated with the client session.

Return Value

If the function succeeds, the return value is the context size for the model. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetModelContextSize** function returns the maximum number of tokens that can be processed by the specified model in a single request. This value is commonly referred to as the model's context size or context window.

The context size includes both the input provided to the model and the output generated in response. This means that the total number of tokens in a request, including the prompt, conversation history, and generated response, must not exceed this limit.

If the combined number of tokens exceeds the model's context size, the request may be rejected by the provider or the response may be truncated. Applications should take this limit into account when constructing prompts or maintaining long conversation histories.

In general, larger context sizes allow a model to retain more information from previous interactions, but may also increase latency and cost depending on the provider and model being used.

The context size returned by this function should be treated as an approximate limit rather than an absolute value. For some providers, particularly locally hosted models, the effective context size may be configured or adjusted outside of the application. In these cases, the reported value serves as a guideline and may not reflect the exact limit enforced by the model at runtime.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEstimateTokenCount, LlmGetModelInfo, LlmGetTokenUsage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetModelFlags Function

```
BOOL WINAPI LlmGetModelFlags(
    HCLIENT hClient,
    LPDWORD lpdwFlags
);
```

The **LlmGetModelFlags** function returns the flags associated with the current model.

Parameters

hClient

A handle to the client session.

lpdwFlags

A pointer to an unsigned integer which may have one or more bit flags set when the function returns. If there are no capabilities defined for the provider, this value will be set to **LLM_MODEL_FLAG_NONE**. This parameter cannot be NULL and the value does not need to be initialized by the caller.

Constant	Description
LLM_MODEL_FLAG_NONE	No model capabilities defined
LLM_MODEL_FLAG_CHAT	Supports conversational text generation
LLM_MODEL_FLAG_SYSTEM_ROLE	Accepts a system role message in the input
LLM_MODEL_FLAG_STREAMING	Supports streaming responses
LLM_MODEL_FLAG_STRUCTURED	Supports structured output (e.g., JSON or schema)
LLM_MODEL_FLAG_VISION	Supports image inputs (multimodal)
LLM_MODEL_FLAG_TOOLS	Supports tool or function calling
LLM_MODEL_FLAG_REASONING	Supports reasoning-oriented generation
LLM_MODEL_FLAG_RESPONSES	Supports the Responses API (unified endpoint)
LLM_MODEL_FLAG_FUNCTIONS	Supports OpenAI-style function calling with schemas
LLM_MODEL_FLAG_AUDIO	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	Supports image generation
LLM_MODEL_FLAG_VIDEO	Supports video generation
LLM_MODEL_FLAG_TRANSCRIPTION	Supports speech-to-text transcription
LLM_MODEL_FLAG_INSTRUCTIONS	Supports a dedicated instruction field outside message lists
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	Uses **max_completion_tokens** instead of **max_tokens** in requests

LLM_MODEL_FLAG_INFERRED	Capabilities are inferred rather than explicitly defined

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetModelFlags** function returns a set of bit flags that describe the capabilities supported by the current model associated with the client session. These flags can be used by applications to determine which features are available and adjust behavior accordingly.

Model capabilities may vary between providers and even between models offered by the same provider. Not all providers return complete capability information when enumerating models. In these cases, some flags may be inferred by the library based on the model name and known characteristics of the provider.

Because capability information may be incomplete or inferred, applications should treat these flags as guidance rather than absolute guarantees of behavior. The provider ultimately determines which features are supported and how they are implemented.

The value returned in the *lpdwFlags* parameter is a bitmask and may contain multiple flags combined using a bitwise OR operation. If no capabilities are defined or available for the model, the value will be set to `LLM_MODEL_FLAG_NONE`. In practice, most supported models provide at least basic conversational capabilities, so a value of zero is uncommon and typically indicates that capability information is not available or could not be determined.

In some cases, capability information may be incomplete or inferred. If necessary, applications can override the detected capabilities using the **LlmSetModelFlags** function. This can be useful when working with providers or models that require specific options which are not automatically detected. In general, applications should rely on the default capability detection unless a specific compatibility issue is encountered.

Some flags may be reserved for internal use or represent provider-specific capabilities. Applications should not rely on undocumented flag values and should only test for defined LLM_MODEL_FLAG constants.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmEnumModels, LlmGetModelContextSize, LlmGetModelInfo, LlmGetModelName, LlmSetModelFlags

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetModelInfo Function

```
BOOL WINAPI LlmGetModelInfo(
   HCLIENT hClient,
   LPCTSTR lpszModelName,
   LPLLM_MODEL_INFO lpModelInfo
);
```

The **LlmGetModelInfo** function returns information about a specific model for the current provider.

Parameters

hClient

A handle to the client session.

lpszModelName

A pointer to a null-terminated string that specifies the name of the model. This parameter may be NULL or an empty string, in which case information about the current model associated with the client session is returned.

lpModelInfo

A pointer to an LLM_MODEL_INFO structure that receives information about the model. This parameter cannot be NULL. The ***dwSize*** member of the structure must be initialized before calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetModelInfo** function retrieves detailed information about a model associated with the provider used by the client session. If a model name is specified, it must be valid for the current provider. If no model name is specified, the function returns information about the model currently associated with the session.

The ***lpModelInfo*** parameter must point to an LLM_MODEL_INFO structure whose ***dwSize*** member has been initialized to `sizeof(LLM_MODEL_INFO)`. This enables the library to validate the structure version and maintain compatibility with future releases.

The information returned by this function includes the model identifier, provider, capability flags, status flags, token limits, context size, and optional metadata such as display name, owner, version, and creation date. Not all providers supply complete metadata for every model, and some values may be empty or unspecified.

Model capability and limit information should be treated as descriptive metadata rather than an absolute guarantee of behavior. Providers may change model availability, limits, or capabilities over time, and locally hosted models may allow some values, such as context size, to be configured outside of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEnumModels, LlmGetModelFlags, LlmGetModelContextSize, LLM_MODEL_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetNextMessage Function

```
BOOL WINAPI LlmGetNextMessage(
   HCLIENT hClient,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **LlmGetNextMessage** function returns the next available message in the client history.

Parameters

hClient

A handle to the client session.

lpMessage

A pointer to an LLM_MESSAGE structure which receives information about the next message in the client history. This parameter may be NULL if the message information is not needed.

lpResponse

A pointer to an LLM_RESPONSE structure which receives information about the response associated with the message. This parameter may be NULL if the response information is not needed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**. When the last message in the history is reached, the error code will be set to `ST_ERROR_END_OF_MESSAGES`.

Remarks

The **LlmGetNextMessage** function is used to enumerate the messages stored in the client history. Each successful call advances the internal enumeration position and returns information about the next message and, if available, the response associated with that message.

If the *lpMessage* or *lpResponse* parameters are specified, the structures must be initialized before calling this function. The dwSize member must be set to the size of the corresponding structure. This enables the function to validate the structure version and safely return the requested information.

Either *lpMessage* or *lpResponse* may be NULL. This is useful when an application only needs to enumerate the original messages or only needs information about the responses. At least one of these parameters should be specified.

To begin enumerating messages, call **LlmGetFirstMessage** to retrieve the first message in the client history. Subsequent messages can then be retrieved by calling **LlmGetNextMessage** until the function returns zero and the last error code is set to ST_ERROR_END_OF_MESSAGES. This condition indicates the enumeration has completed and should not be treated as a failure.

Example

```
// Enumerate the message history and display message details
LLM_MESSAGE clientMessage = { sizeof(LLM_MESSAGE), 0 };
LLM_RESPONSE serverResponse = { sizeof(LLM_RESPONSE), 0 };

if (LlmGetFirstMessage(hClient, 0, &clientMessage, &serverResponse))
{
    do
    {
```

```
        _tprintf(_T("\nMessage %u\n"), clientMessage.dwMessageId);
        _tprintf(_T("  Input Tokens:    %u\n"), clientMessage.nInputTokens);
        _tprintf(_T("  Content Length:  %u\n"), clientMessage.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), clientMessage.lpszContent);

        _tprintf(_T("\nResponse %u\n"), serverResponse.dwMessageId);
        _tprintf(_T("  Elapsed:         %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("  Output Tokens:   %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("  Content Length:  %u\n"), serverResponse.nContentLength);
        _tprintf(_T("  Content:         \"%s\"\n"), serverResponse.lpszContent);
        _tprintf(_T("  Response ID:     %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("  Model Name:      %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("  Status Text:     %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));
    }
    while (LlmGetNextMessage(hClient, &clientMessage, &serverResponse));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHistory, LlmGetFirstMessage, LlmGetMessageCount, LlmSendMessage, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetProviderFlags Function

```
BOOL WINAPI LlmGetProviderFlags(
    DWORD dwProviderId,
    LPDWORD lpdwFlags
);
```

The **LlmGetProviderFlags** function returns the capability flags associated with a provider.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpdwFlags

A pointer to an unsigned integer which receives one or more bit flags when the function returns. If there are no capabilities defined for the provider, this value will be set to LLM_PROVIDER_FLAG_NONE. This parameter cannot be NULL and does not need to be initialized by the caller.

Name	Description
LLM_PROVIDER_FLAG_NONE	No provider requirements specified
LLM_PROVIDER_FLAG_REQUIRE_TLS	Requires a secure TLS (HTTPS) connection
LLM_PROVIDER_FLAG_REQUIRE_AUTH	Requires authentication (e.g., API key or token)
LLM_PROVIDER_FLAG_API_VERSION	Requires an explicit API version parameter
LLM_PROVIDER_FLAG_MODEL_LIST	Supports enumerating available models
LLM_PROVIDER_FLAG_MODEL_INFO	Supports retrieving detailed model information

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetProviderFlags** function retrieves the capability flags associated with the specified provider. These flags describe the features and requirements supported by the provider, such as whether authentication is required or if model enumeration is available.

The value returned in ***lpdwFlags*** may contain one or more LLM_PROVIDER_FLAG values combined with a bitwise OR operation. If no capabilities are defined for the provider, the value will be LLM_PROVIDER_FLAG_NONE.

This function does not require an active client session and can be used to query provider capabilities before establishing a connection. This can be useful when determining how to configure a session or which features are available for a given provider.

If ***dwProviderId*** specifies an invalid provider, the function fails and returns zero.

Example

```
DWORD dwFlags = LLM_PROVIDER_FLAG_NONE;
```

```
if (!LlmGetProviderFlags(dwProviderId, &dwFlags))
{
    _tprintf(_T("Unable to get provider flags\n"));
    return;
}

if (dwFlags & LLM_PROVIDER_FLAG_REQUIRE_AUTH)
{
    _tprintf(_T("Provider requires authentication\n"));
}

if (dwFlags & LLM_PROVIDER_FLAG_MODEL_LIST)
{
    _tprintf(_T("Provider supports model enumeration\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetProviderInfo, LlmGetProviderName, LlmGetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetProviderId Function

```
DWORD WINAPI LlmGetProviderId(
   LPCTSTR lpszProviderName,
   LPCTSTR lpszBaseUrl,
   LPCTSTR lpszModelName
);
```

The **LlmGetProviderId** function returns a provider ID based on a provider name, endpoint URL or model name.

Parameters

lpszProviderName

A null-terminated string which specifies the name of the provider. This parameter may be NULL if you do not wish to match against a provider name. At least one parameter must be specified. If a provider name is specified, it is given precedence over the other parameters.

lpszBaseUrl

A null-terminated string which specifies an endpoint URL. This parameter may be NULL if you do not wish to match against an endpoint URL. At least one parameter must be specified. An endpoint URL will only be used if the function is unable to determine the provider ID from the *lpszProviderName* parameter. If an endpoint URL is specified, it is given precedence over a model name.

lpszModelName

A null-terminated string which specifies a model name. This parameter may be NULL if you do not wish to match against a model name. At least one parameter must be specified. A model name will only be used if the function is unable to determine the provider ID from the *lpszProviderName* or *lpszBaseUrl* parameters.

Return Value

If the function succeeds, the return value is the numeric provider ID. If the function fails, it returns `LLM_PROVIDER_UNKNOWN`. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetProviderId** function attempts to determine the provider ID using the parameters in order of precedence. If a provider name is specified and recognized, that value is considered authoritative and the remaining parameters are ignored. If the provider name is not recognized, the function will attempt to determine the provider from the endpoint URL. If the provider cannot be determined from the endpoint URL, the function will attempt to infer the provider from the model name.

The function performs normalization on the input values before matching them against known providers. Leading and trailing whitespace is ignored, provider names are matched without regard to case and common variations of provider names are recognized.

When matching against a model name, the function identifies the company or organization most commonly associated with the model family. This may not reflect the actual service endpoint hosting the model. For example, a model hosted by a local inference server or aggregate provider may still be identified as the company who originally created the model.

This function only recognizes well-known providers, endpoint URLs and model families. If the provider cannot be determined, the function will return `LLM_PROVIDER_UNKNOWN` and the last error code will be set to `ST_ERROR_PROVIDER_NOT_FOUND`.

The **LlmGetClientProvider** function can be used to return the provider ID associated with a specific client session.

Example

```
DWORD dwProviderId;

// Determine the provider based on a model name
dwProviderId = LlmGetProviderId(NULL, NULL,
                                _T("claude-sonnet-4"));

if (dwProviderId == LLM_PROVIDER_ANTHROPIC)
    _tprintf(_T("Anthropic provider selected\n"));

// Determine the provider using an endpoint URL
dwProviderId = LlmGetProviderId(NULL,
                                _T("https://api.openai.com/v1/"),
                                NULL);

if (dwProviderId == LLM_PROVIDER_OPENAI)
    _tprintf(_T("OpenAI provider selected\n"));
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetClientProvider, LlmGetDefaultModelName, LlmEnumProviders, LlmGetProviderFlags, LlmGetProviderName, LlmValidateModel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetProviderInfo Function

```
BOOL WINAPI LlmGetProviderInfo(
   DWORD dwProviderId,
   LPLLM_PROVIDER_INFO lpProviderInfo
);
```

The **LlmGetProviderInfo** function returns information about the specified service provider type.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpProviderInfo

A pointer to an LLM_PROVIDER_INFO structure which receives information about the specified provider. The dwSize member must be initialized to the size of the structure before calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetProviderInfo** function returns general information about a provider, including its display name, description, default host name, default model and related configuration values.

The *lpProviderInfo* parameter must point to an LLM_PROVIDER_INFO structure initialized by the caller. The dwSize member must be set to the size of the structure. This enables the function to validate the structure version before returning provider information.

This function does not require an active client session. It can be used to query provider information before creating a session or connecting to a provider.

The information returned by this function describes the provider defaults defined by the library. A client session may use different values if the application overrides the provider, host name, base URL, model name or other session settings.

Example

```
DWORD dwProviderId = LLM_PROVIDER_NONE;
TCHAR szBaseUrl[MAX_PATH] = {0};

// Determine the provider ID for the current session and get
// the base URL used for API requests
dwProviderId = LlmGetClientProvider(hClient, szBaseUrl, _countof(szBaseUrl));

if (dwProviderId == LLM_PROVIDER_NONE)
{
    _tprintf(_T("Unable to get provider ID for this session\n"));
    return;
}

// Get the current client options
DWORD dwOptions = LLM_OPTION_NONE;
```

```
if (!LlmGetClientOptions(hClient, &dwOptions))
{
    _tprintf(_T("Unable to get client options for this session\n"));
    return;
}

// Get information about the current provider
LLM_PROVIDER_INFO providerInfo = { sizeof(providerInfo), 0 };

if (!LlmGetProviderInfo(dwProviderId, &providerInfo))
{
    _tprintf(_T("Unable to get provider information\n"));
    return;
}

_tprintf(_T("Connected to %s using %s, the connection is %s\n"),
         providerInfo.szDisplayName,
         szBaseUrl,
         ((dwOptions & LLM_OPTION_SECURE) ? _T("secure") : _T("not secure")));
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetClientProvider, LlmGetProviderFlags, LlmGetProviderName, LlmGetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetProviderName Function

```
INT WINAPI LlmGetProviderName(
   DWORD dwProviderId,
   LPTSTR lpszProviderName,
   INT nMaxLength
);
```

The **LlmGetProviderName** function returns the display name of the specified provider.

Parameters

dwProviderId

Specifies the service provider. This parameter must be one of the **LLM_PROVIDER** constants. Refer to the providers page for a list of supported providers and their default models.

lpszProviderName

A pointer to a string buffer which will contain the display name for the specified provider when the function returns. This parameter cannot be NULL and must be large enough to contain the display name, including the terminating null character. If the buffer is not large enough to contain the entire name, it will be truncated.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. This parameter cannot be zero and the function will fail if an invalid size is provided by the caller.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetProviderName** function returns the display name associated with the specified provider. This is a human-readable string which can be used for logging, status messages or user interface display.

This function is a convenience wrapper which returns the provider display name without requiring the caller to retrieve the full provider information structure using **LlmGetProviderInfo**.

If the buffer specified by ***lpszProviderName*** is not large enough to contain the entire provider name, the string will be truncated. The value returned by the function indicates the number of characters copied to the buffer, not including the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetProviderInfo, LlmGetProviderFlags, LlmGetProviderUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetResponseHeader Function

```
INT WINAPI LlmGetResponseHeader(
   HCLIENT hClient,
   LPCTSTR lpszHeaderName,
   LPTSTR lpszHeaderValue,
   INT nMaxLength
);
```

The **LlmGetResponseHeader** function returns the value of an HTTP header from the server response.

Parameters

hClient

A handle to the client session.

lpszHeaderName

A pointer to a null-terminated string which specifies the name of the HTTP response header. The header names are not case-sensitive. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszHeaderValue

A pointer to a string buffer which will contain the header value when the function returns. If this parameter is NULL, the header value will not be copied and the function will return the length. If the buffer is not large enough to contain the entire header value, it will be truncated.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderValue* string buffer. If *lpszHeaderValue* is NULL, this value should be zero. The maximum length must include room for the terminating null character.

Return Value

If the function succeeds, the return value is the length of the header value in characters, not including the terminating null character. If the header is undefined or the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**. If the header field does not exist, the error code will be `ST_ERROR_HEADER_NOT_FOUND`.

Remarks

The **LlmGetResponseHeader** function returns the value of a response header from the last request made to the service provider.

This function is intended for applications that need lower-level access to provider-specific response headers. Most applications do not need to call this function. Standard response information, such as the HTTP status code, response text, token usage, model name, and elapsed time, is available through the API.

The headers returned by a service provider can vary depending on the provider, model, endpoint, account configuration, and the specific request being made. Applications should not assume that a particular header will always be present unless it is documented by the service provider.

If the header field specified by *lpszHeaderName* was not returned by the server, this function will return zero. If the *lpszHeaderValue* parameter is not NULL, it will contain an empty string when the function returns. The response headers are only available after a request has been sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmResetSession, LlmSendMessage, LlmSetRequestHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetStatusCode Function

```
INT WINAPI LlmGetStatusCode(
    HCLIENT hClient
);
```

The **LlmGetStatusCode** function returns the most recent HTTP status code for a client request.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the HTTP status code. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetStatusCode** function returns the HTTP status code returned by the provider for the most recent request associated with the specified client session. This value reflects the result of the underlying HTTP operation used to communicate with the provider.

HTTP status codes are three-digit numeric values which indicate whether a request was successful or if an error occurred. They may be interpreted using the following general ranges:

Value	Description
100-199	Informational responses. These are rarely used in practice and indicate that the request has been received and processing is continuing.
200-299	Successful responses. The request was successfully processed by the provider. A value of 200 (OK) is typically returned for successful API requests.
300-399	Redirection responses. These indicate that additional action may be required to complete the request. These responses are uncommon when using API endpoints.
400-499	Client errors. The request was invalid or could not be processed by the provider. This may indicate issues such as invalid parameters, missing authentication credentials or rate limiting.
500-599	Server errors. The provider encountered an error while processing the request. These errors are typically transient and may succeed if the request is retried.

This function can be used in conjunction with the return value of a function such as **LlmSendMessage** or **LlmConnect**. The status code may provide additional information about why a request was rejected or processed with an error condition by the provider. For example, if a request fails and the status code is 503, this indicates the service is temporarily unavailable.

HTTP status codes are returned by the provider service and their exact meaning may vary slightly between providers. Some providers may also return additional diagnostic information in the response body which can help identify the specific cause of the error. Here are some common error status codes and what they could mean in the context of the client session:

- **400 Bad Request** - The request was malformed or contained invalid parameters. This usually indicates an internal error in the library where a request was not composed correctly. This could occur if the service provider does not support certain functionality, an invalid option was used,

or an incorrect endpoint was specified.

- **401 Unauthorized** - The API key is missing, invalid, or expired. If you are using an environment variable or Windows credential, check to make sure it is defined and contains the correct value. If you continue to experience this error, check the provider account to ensure the key has not been revoked or replaced.
- **403 Forbidden** - The request was understood, but the provider refused to process it. This can occur if the account does not have permission to use the selected model, endpoint, organization, or project. Verify that the API key has access to the requested resource and that billing or account restrictions are not preventing the request.
- **404 Not Found** - The requested resource could not be found. This commonly indicates an incorrect endpoint URL, deployment name, model name, or provider-specific path. Verify that the provider, base URL, and model name are correct for the account being used.
- **408 Request Timeout** - The provider did not complete the request before the timeout expired. This can occur because of network latency, a temporary service issue, a large prompt, or a model that requires more time to generate a response. Consider increasing the timeout value, reducing the prompt or history size, or retrying the request later.
- **409 Conflict** - The request conflicts with the current state of the service or resource. This is uncommon for simple message requests, but may occur with provider-specific operations, duplicate requests, or incompatible settings. Review the request options and retry the operation after any pending provider-side operation has completed.
- **413 Payload Too Large** - The request body is larger than the provider allows. This usually means the prompt, conversation history, attached content, or generated request JSON exceeds the model or endpoint limit. Reduce the prompt size, clear or shorten the conversation history, or select a model with a larger context size. The **LlmTrimHistory** function can be used to reduce the amount of conversation history included with a prompt.
- **415 Unsupported Media Type** - The provider does not support the request content type or encoding. This can indicate that the request was not submitted as JSON, that the wrong endpoint was used, or that the provider does not support the submitted content format. Verify the endpoint and make sure the request format matches the provider requirements.
- **422 Unprocessable Entity** - The provider accepted the request format, but could not process the request contents. This can occur if a model option, reasoning level, response format, tool definition, or other provider-specific setting is not valid for the selected model. Review the selected model and options, and disable unsupported features.
- **429 Too Many Requests** - The provider rejected the request because a rate limit, usage limit, or account quota was exceeded. Applications should wait before retrying the request, reduce the request rate, and avoid retrying immediately in a tight loop. If the error persists, review the provider account limits, billing status, and usage quotas.
- **500 Internal Server Error** - The provider encountered an unexpected server-side error while processing the request. This usually indicates a temporary problem with the provider rather than the application. Retry the request later, and check the provider status page if the problem continues.
- **502 Bad Gateway** - A gateway or proxy received an invalid response from an upstream service. This is typically a temporary provider-side or network condition. Retry the request after a short delay, and check whether the provider is experiencing a service disruption.

- **503 Service Unavailable** - The service is temporarily unavailable because of maintenance, overload, or resource limits. Retry the request after a delay. If the provider returns a retry interval, the application should wait at least that long before trying again.
- **504 Gateway Timeout** - A gateway or proxy timed out waiting for an upstream service to complete the request. This can occur when the provider is under heavy load or when a response takes too long to generate. Retry the request later, reduce the prompt size, or request a shorter response.

If no request has been made, or if the client has not yet received a response from the provider, the return value may be zero. If the client handle is invalid or a request is currently in progress, the function will return LLM_ERROR (-1). In this case, a status code is not available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnect, LlmGetLastError, LlmSendMessage, LlmTrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetSystemPrompt Function

```
INT WINAPI LlmGetSystemPrompt(
   HCLIENT hClient,
   LPTSTR lpszSystemPrompt,
   INT nMaxLength
);
```

The **LlmGetSystemPrompt** function returns the current system prompt for a client session.

Parameters

hClient

A handle to the client session.

lpszSystemPrompt

A pointer to a string buffer which receives the system prompt for the current session. If this parameter is NULL, the function returns the length of the system prompt without copying any characters. If a buffer is provided, it must be large enough to contain the entire prompt, including the terminating null character, or the function will fail.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. If the ***lpszSystemPrompt*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetSystemPrompt** function retrieves the system prompt associated with the specified client session. The system prompt provides initial instructions to the model and influences how responses are generated for subsequent requests.

To retrieve the system prompt, an application can first call this function with a NULL buffer to determine the required length, then allocate a buffer of sufficient size and call the function again to retrieve the prompt.

If a system prompt has not been defined for the session, the function returns zero and the last error code is set to NO_ERROR. This condition indicates that no prompt is currently set and should not be treated as a failure.

The system prompt may be specified when creating a session using **LlmConnectEx** and the LLM_SESSION structure, or modified at runtime using **LlmSetSystemPrompt**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnectEx, LlmCreateSessionEx, LlmSetSystemPrompt, LLM_SESSION

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetTimeout Function

```
INT WINAPI LlmGetTimeout(
   HCLIENT hClient
);
```

The **LlmGetTimeout** function returns the number of seconds the client will wait for a response from the server.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetTimeout** function returns the current timeout value for the specified client session. This value specifies the maximum amount of time, in seconds, that the client will wait for a response from the provider before the request is aborted.

The timeout applies to network operations performed when communicating with the provider. If the timeout period is exceeded, the request will fail and the last error code will indicate a timeout condition.

The timeout value may be specified when creating a session using **LlmConnectEx** and the LLM_SESSION structure, or modified at runtime using **LlmSetTimeout**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnectEx, LlmGetCacheTime, LlmGetIdleTime, LlmSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetTokenUsage Function

```
BOOL WINAPI LlmGetTokenUsage(
   HCLIENT hClient,
   LPLLM_TOKEN_USAGE lpUsage
);
```

The **LlmGetTokenUsage** function returns token usage statistics for the client session.

Parameters

hClient

A handle to the client session.

lpUsage

A pointer to an LLM_TOKEN_USAGE structure which receives token usage statistics for the client session. The dwSize member must be initialized to the size of the structure before calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetTokenUsage** function returns token usage statistics for the specified client session. This information includes token counts reported by the provider, as well as values maintained by the library for the current conversation history.

The *lpUsage* parameter must point to an LLM_TOKEN_USAGE structure initialized by the caller. The dwSize member must be set to the size of the structure. This enables the function to validate the structure version before returning usage information.

Token usage values are intended to help applications monitor request size, response size and context usage. The exact values reported may vary by provider and model, and should be treated as informational rather than as a guarantee of billing or model behavior.

This function returns the usage information currently associated with the client session. The values are updated as messages are sent and responses are received.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHistory, LlmGetHistorySize, LlmResetSession, LlmSendMessage, LLM_TOKEN_USAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmGetUserAgent Function

```
INT WINAPI LlmGetUserAgent(
   HCLIENT hClient,
   LPTSTR lpszUserAgent,
   INT nMaxLength
);
```

The **LlmGetUserAgent** function returns the current user-agent string used by the client.

Parameters

hClient

A handle to the client session.

lpszUserAgent

A pointer to a string buffer which receives the user-agent string for the current session. If this parameter is NULL, the function returns the length of the user-agent string without copying any characters. If a buffer is provided, it must be large enough to contain the entire string, including the terminating null character, or the function will fail.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the ***lpszUserAgent*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmGetUserAgent** function retrieves the user-agent string currently configured for the client session. This value is included in HTTP requests sent to the provider unless disabled using the LLM_OPTION_NOUSERAGENT option.

If no user-agent string has been explicitly set using **LlmSetUserAgent**, the function may return a default value defined by the library, or no value depending on the current configuration.

To determine the required buffer size, call this function with ***lpszUserAgent*** set to NULL. The return value specifies the number of characters in the string, excluding the terminating null character. If there is no user-agent string defined for the session, the function returns zero and the last error value is cleared (set to NO_ERROR). This condition should not be treated as a failure.

In most cases, the user-agent string will be relatively small. A buffer of 256 characters is typically sufficient for standard user-agent values. If a custom user-agent string is used, a larger buffer (such as 512 characters) may be more appropriate. Applications that require complete flexibility should query the required length and allocate the buffer dynamically.

The user-agent string is typically used by providers for logging or request identification. Developers should avoid including sensitive information in this value.

Example

```
INT cchUserAgent = 0;
```

```
// Query required buffer size
cchUserAgent = LlmGetUserAgent(hClient, NULL, 0);

if (cchUserAgent > 0)
{
    LPTSTR lpszUserAgent = (LPTSTR)malloc((cchUserAgent + 1) * sizeof(TCHAR));

    if (lpszUserAgent != NULL)
    {
        if (LlmGetUserAgent(hClient, lpszUserAgent, cchUserAgent + 1))
        {
            _tprintf(_T("User-Agent: %s\n"), lpszUserAgent);
        }
        else
        {
            DWORD dwError = LlmGetLastError();
            _tprintf(_T("Error retrieving user-agent (0x%08lX)\n"), dwError);
        }

        free(lpszUserAgent);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetClientOptions, LlmGetLastError, LlmSetClientOptions, LlmSetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmImportHistory Function

```
BOOL WINAPI LlmImportHistory(
   HCLIENT hClient,
   LPCTSTR lpszFileName,
   LONG nMessageCount,
   LPLONG lpnImported
);
```

The **LlmImportHistory** function imports a conversation history from a JSON-formatted file.

Parameters

hClient

A handle to the client session.

lpszFileName

A pointer to a null-terminated string which specifies the name of the import file. This parameter cannot be NULL or a zero-length string.

nMessageCount

The number of the most recent messages and responses which should be imported. If this value is zero, all messages in the file are imported.

lpnImported

A pointer to an integer which will receive the number of messages which were imported when the function returns. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmImportHistory** function imports a previously exported conversation history file and restores it as the active conversation history for the client session. Imported messages and responses become part of the current session history and are treated the same as messages generated during the current session.

Importing conversation history replaces the existing history associated with the client session. This function does not merge imported messages with the current conversation history or append them to the existing session. It is not necessary for your application to call **LlmClearHistory** prior to calling this function.

If the *lpszFileName* parameter specifies a directory or device name, the function will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters, they will be removed. The file must exist and the current process must have permission to read the file.

If the *nMessageCount* parameter is non-zero, only the most recent messages and responses in the import file are restored. If the specified count exceeds the number of available messages in the file, all available messages are imported. If the total number of messages to be imported exceeds the current history size, the function will only import the most recent messages up to that limit. For example, if the import file contains 20 messages and the current history size is 15, the most recent 15 messages will be imported.

The import file must contain valid UTF-8 encoded JSON data in a format compatible with the **LlmExportHistory** function. If the file does not appear to contain a valid conversation history, the import operation fails before the current session history is cleared.

After the JSON data has been successfully parsed, the current session history is cleared before the imported messages and responses are added to the client session. If an error occurs while processing imported messages or responses, the previous conversation history cannot be restored and partial import data may remain in the current session.

If the import file contains malformed message or response data, the **LlmGetLastError** function will return `ST_ERROR_INVALID_MESSAGE_FORMAT`.

Because imported history files may contain sensitive or private information, applications should take reasonable precautions to protect them. Applications which allow users to import conversation history should consider warning users before importing untrusted or unknown files.

Example

```
LONG nImported = 0;

// Replace the current conversation history with the
// messages stored in the import file
if (LlmImportHistory(hClient,
                     _T("%USERPROFILE%\\Documents\\conversation.json"),
                     0,
                     &nImported))
{
    _tprintf(_T("Imported %ld messages\n"), nImported);
}
else
{
    DWORD dwError = LlmGetLastError();
    _tprintf(_T("Unable to import conversation history, error 0x%08lX\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHistory, LlmExportHistory, LlmGetMessageCount, LlmSetHistorySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmInitialize Function

```
BOOL WINAPI LlmInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **LlmInitialize** function initializes the library and validates the specified license key.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This function should typically be called once during application startup, before any other library functions are used. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **LlmInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **LlmUninitialize**.

This function dynamically loads system libraries and allocates thread-local storage. If you are calling this library from within another DLL, do not call **LlmInitialize** or **LlmUninitialize** from the **DllMain** function. Doing so can result in deadlocks or access violations. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnect, LlmConnectEx, LlmDisconnect, LlmResetSession, LlmUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmIsBlocking Function

```
BOOL WINAPI LlmIsBlocking(
   HCLIENT hClient
);
```

The **LlmIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero. To get extended error information, call **LlmGetLastError**.

Remarks

A blocking operation may allow the application to be re-entered. For example, the user may press a button while another request is still in progress. If this causes another blocking function to be called for the same client session, the second operation will fail because only one blocking operation can be active at a time.

The **LlmIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action, such as warning the user that they must wait for the current operation to complete.

When a message is sent to the server using the **LlmSendMessage** or **LlmSendMessageEx** functions, the client will block until the message is sent and the server has responded, or until a timeout has occurred. If you send a request to the server in your application's main UI thread, you should present some visual indicator that the application is waiting for the operation to complete. If you need to ensure your application remains responsive while waiting for a response from the server, you can send messages using a background worker thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmCancelRequest, LlmIsConnected, LlmSendMessage, LlmSendMessageEx, LlmSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmIsConnected Function

```
BOOL WINAPI LlmIsConnected(
   HCLIENT hClient
);
```

The **LlmIsConnected** function verifies that the client session remains valid.

Parameters

hClient

A handle to the client session.

Return Value

If the client session is valid and connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmIsConnected** function determines whether there is a logical connection to the service provider and the client has successfully authenticated with the API server. Calling this function is not simply a passive status check. The network connection to the provider may be transient and can be automatically re-established if the client session has been idle. Calling **LlmIsConnected** may cause the library to reconnect and re-authenticate with the server if necessary.

If the client must re-authenticate, it will attempt to reconnect to the server and validate the selected model for the session. If the server reports an error condition, such as an invalid API key, the connection will not be re-established and the function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnect, LlmConnectEx, LlmDisconnect, LlmGetIdleTime, LlmIsBlocking, LlmSetIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmNormalizeText Function

```
INT WINAPI LlmNormalizeText(
   LPCTSTR lpszInput,
   INT cchInput,
   LPTSTR lpszOutput,
   INT nMaxLength
);
```

The **LlmNormalizeText** function normalizes message text by removing extraneous whitespace and applying consistent formatting.

Parameters

lpszInput

A pointer to a null-terminated string which specifies the text which should be normalized. This parameter cannot be NULL.

cchInput

An integer which specifies the length of the input string. If this value is -1 the entire input string up to the terminating null character will be normalized.

lpszOutput

A pointer to a string buffer which will contain the normalized text when the function returns. The output string will always be null-terminated. If the string buffer is not large enough to contain all of the normalized text, the function will fail and the last error code will be set to `ST_ERROR_BUFFER_TOO_SMALL`.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the output string buffer. This value must be large enough to include a terminating null character. If this value is too small, or an invalid length is specified, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied to the output buffer, not including the terminating null character. If the function fails, the return value is zero and the **LlmGetLastError** function will return a non-zero value.

Remarks

The **LlmNormalizeText** function processes the input string, decomposing Unicode characters to their canonical equivalents, replacing confusable homoglyphs with normalized characters, removing combining characters, normalizing end-of-line characters, and trimming leading and trailing whitespace including non-breaking spaces and other special Unicode space characters. Although removing unnecessary whitespace may slightly reduce token usage, the primary benefit is producing cleaner and more consistent input for the model to process.

Line break characters are normalized to use Windows end-of-line conventions. Isolated carriage return or line feed characters, Unicode line and paragraph separators, and form feed characters are converted to CRLF sequences. Multiple consecutive line breaks are reduced to improve consistency.

Normalization can be useful when comparing prompts, checking for empty or whitespace-only input, or applying simple validation rules before a request is submitted. It does not attempt to determine whether the text is safe, appropriate, or semantically valid. Applications should perform any additional validation required for their own use case.

Using this function is optional. Prompt text does not need to be normalized before it is sent to a provider, and using this function is not a required sanitization step. If preserving the exact formatting of the original text is important, such as indentation, tables, or code samples, the application should send the original text instead.

The ANSI version of this function expects the input to be UTF-8 encoded, but will fall back to using the current active code page if invalid UTF-8 characters are encountered in the string. The output from this function will always be valid Unicode text. For applications using the ANSI version of this function, this means the output string will use UTF-8 encoding.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEstimateTokenCount, LlmSendMessage, LlmSendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmResetSession Function

```
BOOL WINAPI LlmResetSession(
    HCLIENT hClient
);
```

The **LlmResetSession** function resets the current client session to its initial state, clearing any existing conversation context.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmResetSession** function resets the client session to its initial state. Any active connection to the service provider is closed, the conversation history is cleared, and token usage statistics for the session are reset.

This function can be used to discard the current conversation context and start a new session using the existing client handle. Configuration settings such as the selected provider, model name, system prompt, and client options are preserved.

If a blocking operation is currently in progress, this function will fail. Applications should call **LlmIsBlocking** to determine if the client is busy before attempting to reset the session.

Because this function clears the conversation history, it also invalidates any string pointers previously returned in LLM_RESPONSE structures or by message history functions. Applications must not use those pointers after this function returns. If the application needs to keep any message or response text, it should make its own copy before resetting the session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmClearHistory, LlmDisconnect, LlmTrimHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSendMessage Function

```
BOOL WINAPI LlmSendMessage(
   HCLIENT hClient,
   LPCTSTR lpszPrompt,
   LPTSTR lpszResponse,
   LPDWORD lpdwLength,
   LPDWORD lpdwMessageId
);
```

The **LlmSendMessage** function sends a prompt to the model and returns the response as a string.

Parameters

hClient

A handle to the client session.

lpszPrompt

A pointer to a null-terminated string which specifies the prompt which should be sent to the model. This parameter cannot be NULL or a zero-length string. If the string consists only of whitespace characters, or contains invalid Unicode or non-text data, this function will fail.

lpszResponse

A pointer to a string buffer which will contain the model's response to the prompt. The response will always be null terminated and if the buffer is not large enough to store the complete response, the response will be truncated. The ***lpdwLength*** parameter must be initialized with the maximum number of characters which can be copied into the buffer or the function will fail.

lpdwLength

A pointer to an unsigned integer which should be initialized to the size of the ***lpszResponse*** string buffer. When the function returns, this value will be updated with the actual number of characters copied into the buffer, not including the terminating null character. This parameter cannot be NULL and its initial value must be greater than zero.

lpdwMessageId

A pointer to an unsigned integer which will contain the ID for the message when the function returns. This parameter may be NULL if the application does not require the message ID.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSendMessage** function provides a simplified interface for sending a single text prompt to the model and receiving the response as a string. This function is intended for applications that do not require structured input or detailed control over message formatting.

This function will not return until the request has completed, an error has occurred, or the operation has timed out. Applications that call this function from a user interface thread should provide visual feedback while waiting for the response, or perform the request on a background thread to maintain responsiveness.

Applications that accept user input should apply appropriate validation, filtering, or normalization before sending requests to the service provider. The **LlmValidateText** function verifies that the

prompt contains valid message text and the **LlmNormalizeText** function can be used to normalize text formatting. These functions do not perform any semantic analysis of the prompt and do not provide protection against malicious or intentionally misleading input.

This function does not validate or restrict the content of the input message. It is the responsibility of the application to ensure that user-provided input is appropriate for its intended use. This includes considering scenarios where input may attempt to override instructions, inject unintended prompts, or otherwise influence the behavior of the model.

The response is written to the buffer specified by ***lpszResponse***. If the buffer is not large enough to contain the complete response, the text will be truncated. The ***lpdwLength*** parameter will always contain the number of characters actually copied to the buffer.

Line breaks in the response text are normalized before being stored in the client history. Line feed and carriage return characters are converted to the standard Windows CRLF end-of-line sequence, ensuring consistent formatting across different providers and applications. This also helps ensure compatibility with Windows common controls and other user interface components.

Some models are capable of performing additional internal reasoning before generating a response. The amount of reasoning performed and whether it is exposed by the provider is model-specific. The library returns the final response intended for the user and does not currently expose model reasoning or intermediate processing steps. This behavior ensures consistent results across providers, regardless of how individual models implement or expose reasoning capabilities.

If the ***lpdwMessageId*** parameter is specified, it will receive a unique identifier for the message within the current client session. Message identifiers are assigned sequentially, but applications should treat them as opaque values. They are only guaranteed to be unique within a single session and should not be compared across different client sessions.

This function automatically adds the message to the conversation history for the session. Subsequent calls will include previous messages as context, subject to any configured history or token limits. To clear the conversation context, call **LlmClearHistory** or **LlmResetSession**.

Applications that require more advanced features, such as structured messages, role-based input, or access to detailed response metadata, should use the **LlmSendMessageEx** function instead.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    HCLIENT hClient = INVALID_CLIENT;
    DWORD dwError = NO_ERROR;

    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    // Establish a connection to OpenAI using the default endpoint
    // and a lightweight model
    hClient = LlmConnect(
        LLM_PROVIDER_OPENAI,
        LLM_OPTION_DEFAULT,
```

```
        LLM_TIMEOUT,
        NULL,                   // Use default base URL
        _T("gpt-5.4-nano"),     // Model name
        _T("%OPENAI_API_KEY%")  // Provider API key
    );

    if (hClient == INVALID_CLIENT)
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        LlmUninitialize();
        return 1;
    }

    TCHAR szResponse[4096] = {0};
    DWORD dwResponseLength = _countof(szResponse); // number of TCHARs
    DWORD dwMessageId = 0;

    // Send a simple message to the model
    if (LlmSendMessage(hClient,
                       _T("What is the capital city of Denmark?"),
                       szResponse,
                       &dwResponseLength,
                       &dwMessageId))
    {
        _tprintf(_T("Response:\n%s\n"), szResponse);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Clean up
    LlmDisconnect(hClient);
    LlmUninitialize();

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHistory, LlmGetMessageById, LlmGetTokenUsage, LlmNormalizeText, LlmResetSession, LlmSendMessageEx, LlmValidateText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSendMessageEx Function

```
BOOL WINAPI LlmSendMessageEx(
   HCLIENT hClient,
   DWORD dwReserved,
   LPLLM_MESSAGE lpMessage,
   LPLLM_RESPONSE lpResponse
);
```

The **LlmSendMessageEx** function sends a structured message to the model and returns a structured response.

Parameters

hClient

A handle to the client session.

dwReserved

A parameter reserved for future use. This value should always be zero.

lpMessage

A pointer to a LLM_MESSAGE structure which specifies the message to be sent to the server. This parameter cannot be NULL. The structure must be initialized correctly, with the ***dwSize*** member set to the size of the structure. Unused members of the structure should be initialized to zero or NULL.

lpResponse

A pointer to a LLM_RESPONSE structure which will be populated with information about the response from the server when the function returns. This parameter cannot be NULL. The structure must be initialized correctly, with the ***dwSize*** member set to the size of the structure and all other members set to a value of zero or NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSendMessageEx** function provides the structured form of **LlmSendMessage**. It should be used by applications that need to specify additional message properties, inspect detailed response metadata, or preserve information such as response identifiers, token usage, status values, elapsed time, and the actual model name returned by the provider.

This function performs a blocking operation. It will not return until the request has completed, an error has occurred, or the operation has timed out. Applications that call this function from a user interface thread should provide visual feedback while waiting for the response, or perform the request on a background thread to maintain responsiveness.

The ***lpMessage*** and ***lpResponse*** structures must be initialized before calling this function. At a minimum, the ***dwSize*** member of each structure must be set to the size of the structure, and all unused members should be initialized to zero or NULL.

If a message identifier is returned in the response structure, it is unique within the current client session. Message identifiers are assigned sequentially by the library, but applications should treat them as opaque values. They are not guaranteed to be globally unique across multiple sessions or between different clients.

This function automatically adds the message and response to the conversation history for the session. Subsequent calls may include previous messages as context, subject to any configured history or token limits. To clear the conversation context, call **LlmClearHistory** or **LlmResetSession**.

Applications that accept user input should apply appropriate validation, filtering, or normalization before sending requests to the service provider. The **LlmValidateText** function verifies that the prompt contains valid message text and the **LlmNormalizeText** function can be used to normalize text formatting. These functions do not perform any semantic analysis of the prompt and do not provide protection against malicious or intentionally misleading input.

Line breaks in the response text are normalized before being stored in the client history. Line feed and carriage return characters are converted to the standard Windows CRLF end-of-line sequence, ensuring consistent formatting across different providers and applications. This also helps ensure compatibility with Windows common controls and other user interface components.

Some models are capable of performing additional internal reasoning before generating a response. The amount of reasoning performed and whether it is exposed by the provider is model-specific. The library returns the final response intended for the user and does not currently expose model reasoning or intermediate processing steps. This behavior ensures consistent results across providers, regardless of how individual models implement or expose reasoning capabilities.

The string members of the LLM_MESSAGE and LLM_RESPONSE structures are managed by the library. When a message is sent, any string pointers provided in the LLM_MESSAGE structure are copied internally, and the application is not required to preserve or maintain those buffers after the function returns.

String pointers returned in the LLM_RESPONSE structure refer to memory managed by the client session. These values remain valid until the message history is cleared or the session is reset or disconnected. Applications should not attempt to free or modify these strings.

The contents of these strings must be treated as read-only. They are owned by the library, and modifying them may result in access violations or other undefined behavior. If your application needs to retain a string beyond the lifetime of the session or modify its contents, it should create its own copy.

Example

```
#include "cstools12.h"
#include <stdio.h>

int _tmain(void)
{
    HCLIENT hClient = INVALID_CLIENT;
    DWORD dwError = NO_ERROR;

    // Initialize the SocketTools library
    if (!LlmInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Initialization failed: %lu\n"), LlmGetLastError());
        return 1;
    }

    // Establish a connection to OpenAI using the default endpoint
    // and a lightweight model
    hClient = LlmConnect(
        LLM_PROVIDER_OPENAI,
        LLM_OPTION_DEFAULT,
        LLM_TIMEOUT,
```

```
        NULL,                   // Use default base URL
        _T("gpt-5.4-nano"),     // Model name
        _T("%OPENAI_API_KEY%")  // Provider API key
    );

    if (hClient == INVALID_CLIENT)
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Unable to connect, error %lu\n"), dwError);
        LlmUninitialize();
        return 1;
    }

    LLM_MESSAGE clientMessage;
    ZeroMemory(&clientMessage, sizeof(LLM_MESSAGE));
    clientMessage.dwSize = sizeof(clientMessage);
    clientMessage.lpszContent = _T("What is the capital city of Denmark?");
    clientMessage.nContentLength = _tcslen(clientMessage.lpszContent);

    LLM_RESPONSE serverResponse;
    ZeroMemory(&serverResponse, sizeof(serverResponse));
    serverResponse.dwSize = sizeof(serverResponse);

    if (LlmSendMessageEx(hClient, 0, &clientMessage, &serverResponse))
    {
        _tprintf(_T("Flags: 0x%08lX\n"), serverResponse.dwFlags);
        _tprintf(_T("Status: %u\n"), serverResponse.dwStatus);
        _tprintf(_T("Elapsed: %u ms\n"), serverResponse.dwElapsed);
        _tprintf(_T("Output Tokens: %u\n"), serverResponse.nOutputTokens);
        _tprintf(_T("Content Length: %u\n"), serverResponse.nContentLength);
        _tprintf(_T("Response ID: %s\n"), (serverResponse.lpszResponseId ?
serverResponse.lpszResponseId : _T("None")));
        _tprintf(_T("Model Name: %s\n"), (serverResponse.lpszModelName ?
serverResponse.lpszModelName : _T("None")));
        _tprintf(_T("Status Text: %s\n"), (serverResponse.lpszStatusText ?
serverResponse.lpszStatusText : _T("None")));

        if (serverResponse.nContentLength > 0)
            _tprintf(_T("\n%s\n"), serverResponse.lpszContent);
    }
    else
    {
        dwError = LlmGetLastError();
        _tprintf(_T("Request failed, error %lu\n"), dwError);
    }

    // Clean up
    LlmDisconnect(hClient);
    LlmUninitialize();

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHistory, LlmGetMessageById, LlmGetTokenUsage, LlmNormalizeText, LlmResetSession, LlmSendMessage, LlmValidateText, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetCacheTime Function

```
BOOL WINAPI LlmSetCacheTime(
   HCLIENT hClient,
   INT nCacheTime
);
```

The **LlmSetCacheTime** function sets the cache lifetime used by the client for model or provider data.

Parameters

hClient

A handle to the client session.

nCacheTime

An integer value which specifies the cache lifetime, in seconds, for provider and model metadata. A value of zero disables caching and forces the client to retrieve this information from the server for each request. Larger values reduce network requests but may result in stale information being used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetCacheTime** function controls how long the client caches provider and model metadata. This includes information such as the list of available models and details about the currently selected model.

Caching reduces the number of network requests required when retrieving model lists or model information, which can improve performance and reduce latency. However, because model availability and capabilities may change over time, cached data may become stale if it is retained for extended periods.

Changing this value will have no effect if the LLM_OPTION_NOCACHE option has been specified.

Setting the cache time to zero disables caching and is equivalent to specifying the LLM_OPTION_NOCACHE option. Note that this differs from specifying a cache time of zero in the LLM_SESSION structure during initialization, where a value of zero indicates that a default cache lifetime will be used.

The cache time is specified in seconds. If a value greater than or equal to 1000 is specified, it is assumed that the value may have been provided in milliseconds, and it will be automatically scaled to seconds. The specified cache time is clamped to an internal minimum and maximum value if it falls outside the supported range. The default cache time is one hour.

In most cases, the default cache time provides a good balance between performance and accuracy. Applications that frequently query model information or require real-time updates may choose to reduce or disable caching.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmEnumModels, LlmGetCacheTime, LlmGetModelInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetClientOptions Function

```
BOOL WINAPI LlmSetClientOptions(
    HCLIENT hClient,
    DWORD dwOptions
);
```

The **LlmSetClientOptions** function changes the current client option flags.

Parameters

hClient

A handle to the client session.

dwOptions

An unsigned integer that specifies one or more options.

Value	Description
LLM_OPTION_NOCACHE	Disables provider and model metadata caching for the client session. When this option is specified, the library will not reuse cached metadata, and provider or model information will be retrieved from the server each time it is required. This may be useful when testing or when current data is required.
LLM_OPTION_KEEPALIVE	Enables persistent HTTP connections using the keep-alive mechanism. When enabled, the underlying HTTP connection may be reused for multiple requests, reducing connection overhead and improving performance. If this option is not specified, the connection may be closed after each request.
LLM_OPTION_PROXY	Specifies that the connection should be made using the current Windows proxy server configuration. If the user does not have a proxy server configuration, this option is ignored. This option is typically used in environments where direct internet access is restricted.
LLM_OPTION_NOUSERAGENT	Prevents the library from sending a User-Agent header with requests to the provider. By default, a User-Agent string is included to identify the client. This option may be used when a provider does not require a User-Agent header.
LLM_OPTION_SECURE	Forces the use of a secure HTTPS connection when communicating with the provider. If the specified base URL or provider default would normally use an insecure protocol, it will be upgraded to a secure connection. Connections to providers such as OpenAI, Google and Anthropic will always be secure.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To

get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetClientOptions** function replaces the current option flags for the client session. These flags control how the client establishes connections and sends requests to the service provider.

Applications should use care when changing options that affect the underlying connection, such as LLM_OPTION_SECURE or LLM_OPTION_PROXY. Changing these options may require the client to establish a new connection using different settings.

The LLM_OPTION_SECURE option forces the client to use HTTPS when communicating with the provider. This is appropriate for public API providers, but may not be supported by locally hosted providers such as LM Studio or Ollama. Attempting to force a secure connection to a provider that does not support HTTPS will cause the connection to fail.

The LLM_OPTION_PROXY option causes the client to use the configured proxy settings when connecting to the provider. This may be required in managed or corporate network environments, but should not be enabled unless proxy support is actually needed.

The LLM_OPTION_NOUSERAGENT option does not make requests anonymous. The provider may still identify the client based on authentication credentials, IP address information, request patterns, or other HTTP headers included with the request. For more information, refer to the **LlmSetUserAgent** method.

Setting or clearing the LLM_OPTION_NOCACHE option changes whether provider and model metadata may be cached. Applications can also use the **LlmSetCacheTime** function to control caching behavior in seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnect, LlmConnectEx, LlmGetCacheTime, LlmGetClientOptions, LlmGetUserAgent, LlmSetCacheTime, LlmSetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetHistorySize Function

```
BOOL WINAPI LlmSetHistorySize(
    HCLIENT hClient,
    LONG nMaxMessages
);
```

The **LlmSetHistorySize** function sets the maximum number of message exchanges retained in the conversation history.

Parameters

hClient

A handle to the client session

nMaxMessages

An integer value which specifies the maximum number of messages that will be retained in the conversation history for the client session. Older messages will be discarded automatically as new messages are added once this limit is reached. This value cannot be less than zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetHistorySize** function controls the number of messages retained in the conversation history for a client session. This history is used to provide context for subsequent requests to the model.

When the number of messages exceeds the specified limit, the oldest messages are removed automatically. This ensures that the total number of stored messages does not exceed the configured maximum.

Reducing the history size can help limit memory usage and reduce the amount of context sent with each request, which may improve performance and reduce token usage. However, limiting the history size may also reduce the model's ability to maintain context across multiple interactions.

This setting works in conjunction with token limits imposed by the selected model. Even if the maximum number of messages has not been reached, older messages may be excluded from a request if including them would exceed the model's context window.

If *nMaxMessages* has a value of zero, conversation history is effectively disabled. In this mode, only the current message and response are retained for the duration of the request, and no prior messages are included as context for subsequent requests. Disabling conversation history may reduce the model's ability to maintain context across multiple interactions and can impact the quality or relevance of responses. Functions such as **LlmGetFirstMessage** and **LlmGetNextMessage** will fail with the last error code set to ST_ERROR_NO_MESSAGE_HISTORY.

Disabling conversation history does not provide additional security or privacy. It only affects whether previous messages are retained and automatically included in subsequent requests. The content of each request is still transmitted to the service provider and may be processed or stored in accordance with that provider's policies. Applications that require secure handling of sensitive information should avoid including confidential data in requests and review the data handling practices of the selected service provider.

Even when message history is disabled, the model may still produce responses that appear contextually relevant within a single interaction. This is because the current request can include sufficient context for the model to generate an appropriate response. However, without retaining previous messages, the model does not have access to prior exchanges in the conversation, and continuity across multiple requests may be limited. In most cases, retaining a moderate history size provides better results for conversational applications.

The default history size is 100 message exchanges.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmClearHistory, LlmGetHistorySize, LlmResetSession, LlmSendMessage, LlmSendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetIdleTime Function

```
BOOL WINAPI LlmSetIdleTime(
   HCLIENT hClient,
   INT nIdleTime
);
```

The **LlmSetIdleTime** function sets the idle timeout period for the client session.

Parameters

hClient

A handle to the client session.

nIdleTime

An integer value which specifies the idle timeout period in seconds. A value of zero specifies a reasonable default value should be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetIdleTime** function sets the idle timeout period, in seconds, for the specified client session. This value determines how long the library will retain an open connection between requests before automatically closing it and establishing a new connection when needed.

Setting a lower idle time may result in more frequent reconnections, which can increase overhead but may be desirable in environments where connections are short-lived. Higher values allow connections to be reused for longer periods, which can improve performance when making repeated requests.

Changing this value is only meaningful when the LLM_OPTION_KEEPALIVE option is enabled. If keep-alive is disabled for the session, the connection will not be reused regardless of the idle time setting. If the ***nIdleTime*** parameter is set to zero, a new connection is established for each request. In most cases, a default idle time of 15 seconds provides a good balance between performance and compatibility.

This setting applies only to the client. It cannot force a server to maintain a persistent connection for a longer period of time. If a connection is closed due to inactivity, the library will automatically reconnect as needed. This behavior is transparent to the application and does not require any additional function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmIsConnected, LlmGetIdleTime, LlmSetCacheTime, LlmSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetLastError Function

```
VOID WINAPI LlmSetLastError(
   DWORD dwErrorCode
);
```

The **LlmSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or LLM_ERROR. Those functions which call **LlmSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **LlmGetLastError** function. The use of **LlmGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetErrorString, LlmGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetModelFlags Function

```
BOOL WINAPI LlmSetModelFlags(
   HCLIENT hClient,
   DWORD dwModelFlags
);
```

The **LlmSetModelFlags** function sets model behavior flags for a client session.

Parameters

hClient

A handle to the client session.

dwModelFlags

An unsigned integer which specifies one or more capability flags for the selected model. This value cannot be zero.

Name	Description
LLM_MODEL_FLAG_CHAT	Supports conversational text generation
LLM_MODEL_FLAG_SYSTEM_ROLE	Accepts a system role message in the input
LLM_MODEL_FLAG_STREAMING	Supports streaming responses
LLM_MODEL_FLAG_STRUCTURED	Supports structured output (e.g., JSON or schema)
LLM_MODEL_FLAG_VISION	Supports image inputs (multimodal)
LLM_MODEL_FLAG_TOOLS	Supports tool or function calling
LLM_MODEL_FLAG_REASONING	Supports reasoning-oriented generation
LLM_MODEL_FLAG_RESPONSES	Supports the Responses API (unified endpoint)
LLM_MODEL_FLAG_FUNCTIONS	Supports OpenAI-style function calling with schemas
LLM_MODEL_FLAG_AUDIO	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	Supports image generation
LLM_MODEL_FLAG_VIDEO	Supports video generation
LLM_MODEL_FLAG_TRANSCRIPTION	Supports speech-to-text transcription
LLM_MODEL_FLAG_INSTRUCTIONS	Supports a dedicated instruction field outside message lists
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	Uses `max_completion_tokens` instead of `max_tokens` in requests

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To

get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetModelFlags** function overrides the capability flags associated with the currently selected model. These flags determine how the client formats requests and interprets responses for the model.

Model capability flags are normally determined automatically by the library based on the selected provider and model. This function is primarily intended for advanced scenarios where the default behavior must be adjusted, such as when working with custom providers, locally hosted models, or when a model's capabilities have changed and are not yet reflected in the library.

The specified flags must include `LLM_MODEL_FLAG_CHAT`, as the API requires a chat-based interface for sending and receiving messages. Specifying flags that are not supported by the selected model may result in request failures or undefined behavior.

Changing model flags can affect how requests are constructed, such as which parameters are used, how input is formatted, and which features are enabled. Applications should only modify these flags if they fully understand the capabilities of the target model.

If you enable a capability that is not supported by the model or provider, this will often result in an error condition and may cause subsequent requests to fail. For example, if you attempt to force the use of a system prompt by setting the LLM_MODEL_FLAG_SYSTEM_ROLE flag, this can cause unexpected failures on models hosted locally using an application like LM Studio.

Changing the current model using the **LlmSetModelName** function will reset the model flags for the session back to their default values for the new model.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmGetModelInfo, LlmGetModelFlags, LlmGetModelName, LlmSetModelName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetRequestHeader Function

```
BOOL WINAPI LlmSetRequestHeader(
   HCLIENT hClient,
   LPCTSTR lpszHeaderName,
   LPTSTR lpszHeaderValue
);
```

The **LlmSetRequestHeader** function sets the value of a custom HTTP header issued with subsequent requests.

Parameters

hClient

A handle to the client session.

lpszHeaderName

A pointer to a null-terminated string which specifies the name of the HTTP header field. Header names may only contain ASCII letters, numbers and a restricted subset of punctuation characters. Header names cannot contain Unicode or ANSI multi-byte character sequences.

lpszHeaderValue

A pointer to a null-terminated string which specifies the value of the HTTP header. Header values cannot contain carriage-return or linefeed characters. If the value contains Unicode characters, they will be UTF-8 encoded. If this parameter or NULL or a zero-length string, the header field is deleted from the request headers.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**. If the header name contains illegal characters or cannot be changed (e.g., attempting to change the `Authorization` header), the function will return zero and the last error code will be set to `ST_ERROR_INVALID_HEADER_NAME`. If the header value contains illegal characters, the last error code will be set to `ST_ERROR_INVALID_HEADER_VALUE`.

Remarks

The **LlmSetRequestHeader** function assigns a custom HTTP request header that will be included with subsequent requests made using the current client session. If the specified header already exists in the custom header list, its value will be replaced. If the header value is an empty string, the header will be removed.

This function is intended for advanced applications that require lower-level access to provider-specific functionality. Most applications should not need to use custom request headers. Some headers are managed internally by the library and cannot be overridden using this function. This includes headers required for authentication, content formatting, message length, host identification, and other protocol-specific functionality required by the current provider.

Examples of headers that are managed internally by the library include:

- `Authorization`
- `Content-Length`
- `Content-Type`
- `Host`
- `User-Agent`

Custom headers remain associated with the current client session until they are removed using the **LlmClearHeaders** function, the **LlmResetSession** function is called, or the connection is closed using the **LlmDisconnect** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHeaders, LlmDisconnect, LlmGetResponseHeader, LlmResetSession, LlmSendMessage, LlmSetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetSystemPrompt Function

```
BOOL WINAPI LlmSetSystemPrompt(
   HCLIENT hClient,
   LPCTSTR lpszSystemPrompt
);
```

The **LlmSetSystemPrompt** function changes the current system prompt for a client session.

Parameters

hClient

A handle to the client session.

lpszSystemPrompt

A pointer to a string buffer which will contain the new system prompt for the current session. If this parameter is NULL or an empty string, a system prompt will not be included with new requests.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

A system prompt provides instructions that are included with each request sent to the model. It can be used to define the expected role, tone, response style, formatting rules, or other application-specific guidance that should apply to the session.

For example, an application might use a system prompt to request short responses, avoid Markdown formatting, return structured text, or answer only within the context of a specific feature or workflow.

Because conversation history is included with each request, messages that were generated using a previous system prompt may still influence subsequent responses. Changing or clearing the system prompt does not retroactively modify existing messages in the history. If the new prompt represents a significant change in behavior, applications may want to clear the history to avoid unintended interactions between the previous and current instructions.

If the new prompt represents a significant change in behavior, applications may want to call **LlmClearHistory** before sending additional messages.

The system prompt should be treated as guidance, not a guarantee. Different providers and models may interpret system instructions differently, and user messages may still influence the response. If ***lpszSystemPrompt*** is NULL or points to an empty string, the current system prompt is cleared and no system prompt will be included with new requests.

Support for system prompts may vary between providers and models. Some providers treat the system prompt as a distinct message role, while others incorporate it as part of a separate instruction field. The library will map the system prompt to the appropriate format for the selected provider, but the exact behavior and influence on the response may differ. Applications should test system prompt behavior with their selected provider to ensure it produces the desired results.

Note that some providers or models will not provide any support for a system role. This is more common with locally hosted models using applications like LM Studio or Ollama. To determine if a model supports system prompts, use the **LlmGetModelFlags** function and check for the LLM_MODEL_FLAG_SYSTEM_ROLE flag.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmClearHistory, LlmGetModelFlags, LlmGetSystemPrompt, LlmResetSession, LlmSendMessage, LlmSendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetTimeout Function

```
BOOL WINAPI LlmSetTimeout(
   HCLIENT hClient,
   UINT nTimeout
);
```

The **LlmSetTimeout** function sets the number of seconds the client will wait for a blocking operation to complete.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetTimeout** function sets the timeout period, in seconds, for blocking operations such as sending a message to the server. If a request does not complete within the specified time, the operation will fail and return a timeout error.

The minimum timeout period is 5 seconds and the maximum timeout period is 300 seconds. If a value outside this range is specified, it will be clamped to the nearest supported value. Specifying a value of zero resets the timeout to the default value of 60 seconds.

The appropriate timeout value may vary depending on the selected provider, model, and network conditions. Larger or more complex models may require additional time to generate a response. Locally hosted models may also require longer timeouts depending on the available hardware resources, such as CPU or GPU performance.

The maximum timeout period is intended to prevent applications from waiting indefinitely for a response. If an operation consistently requires several minutes to complete, applications should consider performing the request on a background thread or using a smaller or faster model.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnect, LlmGetTimeout, LlmSendMessage, LlmSendMessageEx, LlmSetCacheTime, LlmSetIdleTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmSetUserAgent Function

```
BOOL WINAPI LlmSetUserAgent(
   HCLIENT hClient,
   LPCTSTR lpszUserAgent
);
```

The **LlmSetUserAgent** function changes the current user-agent string used by the client.

Parameters

hClient

A handle to the client session.

lpszUserAgent

A pointer to a string buffer which will contain the new user-agent string for the current session. If this parameter is NULL or a zero-length string, the function will reset the user-agent string to a default value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmSetUserAgent** function changes the user-agent string used by the client session. The specified string is trimmed to remove leading and trailing whitespace and will be used the next time the client establishes a connection to the provider.

If *lpszUserAgent* is NULL or points to an empty string, the current user-agent string is cleared and a new default value is generated for the session.

The default user-agent string uses a format similar to Chromium-based browsers and includes information about the current Windows platform and the SocketTools version. If an application name was specified when the session was created, it may also be included in the default value.

For example, a default user-agent string may look similar to the following:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/134.0.0.0 Safari/537.36
SocketTools/12.0
```

Applications normally do not need to change the user-agent string. However, a custom value may be useful when an application needs to identify itself to a provider, gateway, proxy, or logging system.

Developers should avoid including sensitive information, such as API keys, account identifiers, or user-specific data, in the user-agent string.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetClientOptions, LlmGetUserAgent, LlmSetClientOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmTrimHistory Function

```
LONG WINAPI LlmTrimHistory(
   HCLIENT hClient,
   LONG nMessageCount,
   LONG nMaxTokens
);
```

The **LlmTrimHistory** function removes older messages from the conversation history for the current client session.

Parameters

hClient

A handle to the client session.

nMessageCount

The number of messages to remove from the beginning of the conversation history. If this value is zero, messages are only removed if required by the ***nMaxTokens*** parameter. If this value is equal to or greater than the number of messages in the history, the entire history is cleared.

nMaxTokens

The maximum number of input tokens that should remain in the conversation history. If this value is zero, token usage is not checked. If the current history exceeds this value, additional older messages may be deleted until the total input token count is below the specified limit.

Return Value

If the function succeeds, the return value is the number of messages deleted from the conversation history. If the function fails, the return value is LLM_ERROR. To get extended error information, call **LlmGetLastError**.

If no messages are available to delete, the return value is zero.

Remarks

The **LlmTrimHistory** function removes messages from the beginning of the conversation history. This can be used to reduce memory usage, discard older context, or limit how much previous conversation text is included with future requests.

The ***nMessageCount*** parameter specifies the minimum number of older messages to remove. If ***nMaxTokens*** is greater than zero, the function also checks the total number of input tokens currently stored in the message history. If the token count exceeds the specified limit, additional older messages will be removed until the total input token count is below the specified threshold.

When both parameters are specified, the function may remove more messages than requested by ***nMessageCount*** to reduce the token count below the limit specified by ***nMaxTokens***. The return value always indicates the actual number of messages removed.

If the number of messages to remove is equal to or greater than the number of messages currently stored, the entire conversation history is cleared. This is equivalent to calling **LlmClearHistory**.

This function only removes the local message and response history maintained by the client session. It does not change the current provider, model, system prompt, connection settings or cached provider metadata.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmClearHistory, LlmGetHistorySize, LlmSetHistorySize, LlmGetFirstMessage, LlmGetNextMessage, LlmGetTokenUsage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmUninitialize Function

```
VOID WINAPI LlmUninitialize();
```

The **LlmUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application must call **LlmInitialize** successfully before calling any other library functions. When it has finished using the library, the application should call **LlmUninitialize** to release resources allocated by the library. If this function is not called explicitly, the library will release its resources when the application terminates.

Initialization is reference counted. Each successful call to **LlmInitialize** increments the library usage count, and each call to **LlmUninitialize** decrements it. The library is not fully uninitialized until the usage count reaches zero. When the usage count reaches zero, **LlmUninitialize** will terminate all active client sessions, close any open connections, and free memory associated with the library. Any pending blocking operations are canceled and will not complete.

This function should not be called while other threads are actively using the library. Applications should ensure that all operations have completed and that no additional calls will be made before uninitializing the library.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib

See Also

LlmConnect, LlmConnectEx, LlmDisconnect, LlmInitialize, LlmResetSession

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmValidateModel Function

```
BOOL WINAPI LlmValidateModel(
   HCLIENT hClient,
   LPCTSTR lpszModelName
);
```

The **LlmValidateModel** function validates that a model name is recognized or supported by the current provider.

Parameters

hClient

A handle to the client session.

lpszModelName

A null-terminated string which specifies the name of the model to validate. This parameter cannot be NULL or a zero-length string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmValidateModel** function verifies that a model name appears to be valid for the current session. Validation may be performed using locally available information or by querying the provider for a list of supported models. If the provider supports returning model information, this function may perform a lookup to confirm that the specified model name is recognized. This may require a network request and can cause the function to block while the model list is retrieved.

If the provider does not support returning model information, the function only performs basic validation of the model name. For example, it will check the name to ensure it does not contain any illegal characters and uses a format that is common for most service providers. This function is most commonly used with applications to validate user input.

A successful return value from this function only indicates that the model name is syntactically valid. It does not guarantee that the selected service provider will accept or allow the model to be used. A provider may reject a valid model name for several reasons, including the model being unavailable for the current account, restricted to specific subscription tiers, region-limited, deprecated, disabled by administrative policy, or not supported by the endpoint being used. Providers may also change model availability over time without notice. Applications should be prepared to handle errors returned when establishing a session or sending a request, even if the model name was validated successfully.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEnumModels, LlmGetModelInfo, LlmSetModelFlags, LlmSetModelName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LlmValidateText Function

```
BOOL WINAPI LlmValidateText(
   LPCTSTR lpszMessageText,
   INT cchMessageText
);
```

The **LlmValidateText** function performs basic validation on text before it is submitted to a model.

Parameters

lpszMessageText

A pointer to the message text to validate. This parameter cannot be NULL.

cchMessageText

The number of characters in the message text, excluding the terminating null character. If this value is -1, the message text must be null-terminated and the entire string is checked.

Return Value

If the message text appears to be valid, the function returns non-zero. If the message text is empty, contains invalid characters, or appears to contain non-text data, the function returns zero. To get extended error information, call **LlmGetLastError**.

Remarks

The **LlmValidateText** function is intended to help applications detect invalid or malformed input before sending a request to a model. It checks for common problems such as empty strings, text that consists only of spaces or line breaks, illegal Unicode code points, non-printable characters, and input that appears to contain binary data. The message text cannot contain embedded null characters.

Applications that accept messages from users or other external sources should validate the text before submitting it to a model. This can help detect malformed or invalid input before a request is sent, reducing the chance of unexpected results or incurring unnecessary usage costs.

This function does not evaluate the quality, safety, or semantic meaning of the message. Although this function is most commonly used to check prompts sent to a model, it can be used to validate any type of text to ensure it is not malformed and contains printable characters.

There are two versions of this function. The wide character version of this function (**LlmValidateTextW**) validates the message text as UTF-16 Unicode. The ANSI version (**LlmValidateTextA**) first attempts to interpret the input as UTF-8 encoded text. If the input is not valid UTF-8, it attempts to convert the text to Unicode using the active code page and the resulting Unicode text is validated. The generic function name **LlmValidateText** automatically resolves to the ANSI or Unicode version based on whether the application is compiled with Unicode support.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csllmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEstimateTokenCount, LlmNormalizeText, LlmSendMessage, LlmSendMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Data Structures

- LLM_MESSAGE
- LLM_MODEL_INFO
- LLM_PROVIDER_INFO
- LLM_RESPONSE
- LLM_SESSION
- LLM_TOKEN_USAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_MESSAGE

The **LLM_MESSAGE** structure represents a message sent from the client to the LLM server.

```
typedef struct _LLM_MESSAGE
{
    DWORD   dwSize;
    DWORD   dwRoleId;
    DWORD   dwFlags;
    DWORD   dwReasoningLevel;
    DWORD   dwMessageId;
    DWORD   dwReserved;
    TIME_T  tTimestamp;
    LONG    nMaxTokens;
    LONG    nInputTokens;
    LONG    nContentLength;
    LPCTSTR lpszContent;
} LLM_MESSAGE, *LPLLM_MESSAGE;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_MESSAGE)` before the structure is used.

dwRoleId

The role associated with the message, such as user, assistant or system. This member should be one of the following values:

Constant	Description
LLM_ROLE_UNKNOWN	Unknown or invalid role. This value is typically used when a message has not been initialized correctly or could not be classified.
LLM_ROLE_USER	A message originating from the end-user. This is the most common role and represents input provided by the application or user interface.
LLM_ROLE_ASSISTANT	A response generated by the language model. Messages with this role are typically returned by the library as part of the conversation history.
LLM_ROLE_SYSTEM	A system-level instruction that defines how the model should behave. This is typically used to set context, tone, or constraints for the session before user messages are processed.
LLM_ROLE_FUNCTION	A legacy role used for function call messages in older APIs. This role is retained for compatibility but is generally not used in new implementations.
LLM_ROLE_TOOL	A message generated by an external tool or function invoked by the model. This role is used when integrating tool or function-calling workflows.
LLM_ROLE_TOOLUSER	A user message that is specifically intended for processing by a tool or function, rather than direct model interpretation.

LLM_ROLE_DEFAULT	The default role assigned to messages. This is equivalent to LLM_ROLE_USER.

dwFlags

A value reserved for future use. This member should always be set to zero or `LLM_MESSAGE_FLAG_NONE`.

dwReasoningLevel

The reasoning effort level associated with the message. This member may be set to one of the `LLM_REASONING` constants to control how much reasoning the model should apply when generating a response. A value of `LLM_REASONING_DEFAULT` uses the provider or model default behavior.

dwMessageId

An identifier assigned to the message. For application-supplied input messages, this value is initially zero. When a message has been added to a session or returned from the message history, this member may contain a unique identifier assigned by the library.

dwReserved

A reserved value for future use. This member should always be zero.

tTimestamp

The timestamp associated with the message. For application-supplied input messages, this value is typically zero. When available, this member contains the time the message was created.

nMaxTokens

The maximum number of output tokens associated with the message. This value is primarily used when submitting a request and may be zero to indicate the session or provider default should be used.

nInputTokens

The number of input tokens associated with the message content. This value is typically provided by the library when token usage information is available.

nContentLength

The length of the message content, in characters, not including the terminating null character. If this value is zero, the content length may be determined from the string.

lpszContent

A pointer to a null-terminated string which contains the message text. This member may be NULL if the message consists only of media content.

Remarks

This structure is used to describe a single message in a conversation with a language model. Messages are typically assigned a role using one of the constants, such as `LLM_ROLE_USER` for user input, `LLM_ROLE_ASSISTANT` for a model response, or `LLM_ROLE_SYSTEM` for an instruction message that defines the behavior of the session. In most applications, those roles are required. Other roles are used for advanced scenarios such as tool integration or legacy compatibility.

The **LLM_MESSAGE** structure is used for both input and output operations. When passed to functions such as **LlmSendMessageEx**, it defines a message that will be submitted to the language model. In this case, the application typically initializes only the required members, such as ***dwSize***, ***dwRoleId***, and the content or media fields. When used with functions such as **LlmGetFirstMessage** and **LlmGetNextMessage**, this structure is populated by the library to return messages from the current session history. In this case, additional members such as the message identifier, timestamp, and token

counts may contain values assigned by the library. Applications should treat values returned by the library as read-only unless otherwise documented.

The *lpszContent* member specifies the text associated with the message. For the ANSI version of this structure, this string must be UTF-8 encoded. In the Unicode version, the string is specified as UTF-16 using a wide character pointer.

The *nContentLength* member specifies the content length in characters rather than bytes. If this value is zero, the library may determine the length from the null-terminated string. Applications should not assume that token counts are identical to character counts. The *nInputTokens* member, when available, reflects the tokenized size of the message content as determined by the model or provider.

The *dwReasoningLevel* member controls the reasoning effort requested for the message. Higher reasoning levels may improve the quality, accuracy, or depth of responses, particularly when using reasoning-capable models, but they can also increase response time and token usage costs. Lower reasoning levels generally favor faster responses and reduced usage costs. Not all providers or models support configurable reasoning levels, and unsupported values may be ignored by the provider. If this member is set to `LLM_REASONING_DEFAULT`, the library will use the default reasoning behavior for the current provider or model.

The *tTimestamp* member is a standard 64-bit Unix time value and can be used interchangeably with functions in the C standard library such as **gmtime** or **localtime**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps can be converted into a local date and time using the **LlmConvertTimestamp** function.

Unless otherwise noted, any reserved members should be set to zero before calling a function which accepts this structure. Applications should also avoid modifying structure members that are returned by the library unless the documentation for a specific function indicates that doing so is supported.

The *lpszContent* member of this structure is managed by the library. Applications should treat this value as read-only and must not attempt to modify the contents of the string. If the application needs to retain or modify this data, it should create its own copy.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmSendMessageEx LlmGetMessage, LlmGetMessageById, LlmGetFirstMessage, LlmGetNextMessage, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_MODEL_INFO

The **LLM_MODEL_INFO** structure contains descriptive information and capability data for a language model.

```
typedef struct _LLM_MODEL_INFO
{
    DWORD    dwSize;
    DWORD    dwProviderId;
    DWORD    dwFlags;
    DWORD    dwStatus;
    DWORD    dwReserved;
    LONG     nMaxTokens;
    LONG     nContextSize;
    FLOAT    fTemperature;
    FLOAT    fMaxTemperature;
    FILETIME ftCreated;
    TCHAR    szName[LLM_MODEL_NAMELEN];
    TCHAR    szDisplayName[LLM_MODEL_NAMELEN];
    TCHAR    szOwner[LLM_MODEL_NAMELEN];
    TCHAR    szVersion[LLM_MODEL_VERLEN];
    TCHAR    szCreated[LLM_MODEL_DATELEN];
} LLM_MODEL_INFO, *LPLLM_MODEL_INFO;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_MODEL_INFO)` before the structure is used.

dwProviderId

The provider associated with the model. This member should be one of the LLM_PROVIDER values.

dwFlags

One or more capability flags associated with the model. This member may contain a combination of LLM_MODEL_FLAG values.

dwStatus

One or more status flags associated with the model. This member may contain a combination of LLM_MODEL_STATUS values.

dwReserved

A reserved value for future use. This member should always be zero.

nMaxTokens

The maximum number of tokens the model can generate in response to a request. A value of zero indicates that this limit is not specified.

nContextSize

The maximum context window size for the model, in tokens. If this value is zero, the context size is unspecified or the provider does not distinguish it from ***nMaxTokens***.

fTemperature

The default temperature value associated with the model. A value of zero may indicate the value is not specified.

fMaxTemperature

The maximum supported temperature value for the model. A value of zero indicates that this limit is not specified.

ftCreated

The creation date and time associated with the model, expressed as a FILETIME value. If this member is zero, the creation time is not available.

szName

A null-terminated string which specifies the model identifier used by the provider, such as `gpt-5` or `claude-4-sonnet`.

szDisplayName

A null-terminated string which specifies an optional display name for the model. This value is intended for presentation to the user and may be an empty string.

szOwner

A null-terminated string which specifies the optional owner, vendor, or organization associated with the model. This value may be an empty string.

szVersion

A null-terminated string which specifies the optional version identifier associated with the model. If this information is not available, the value will be an empty string.

szCreated

A null-terminated string which specifies the optional model creation date in text form. If this information is not available, the value will be an empty string.

Remarks

This structure is used to return information about a model that is available through a specific provider. It is typically populated by the library when enumerating models or retrieving detailed information about a specific model. Applications should treat the contents of this structure as descriptive metadata and should not assume that every member will contain a meaningful value for every model.

The ***dwProviderId*** member identifies which service or platform supplies the model, using one of the `LLM_PROVIDER` constants. The ***dwFlags*** member describes the capabilities supported by the model, such as conversational input, structured output, reasoning, vision, tool calling, or audio support. The ***dwStatus*** member indicates the current state of the model, such as whether it is generally available, in preview, or deprecated. These values are defined by the library using the `LLM_MODEL_FLAG` and `LLM_MODEL_STATUS` constants.

The ***nMaxTokens*** value represents the maximum output token limit reported for the model. This limit only applies to generated content and may be lower than the model's total context window. The client can further restrict the length of generated responses if necessary. If this value is zero, it means the provider does not publish information about the maximum number of output tokens supported for a response. Internally, providers often impose generation limits that are lower than the total context size for a request, even if those limits are not publicly documented.

The ***nContextSize*** value represents the maximum number of tokens the model can process in a single request, including the system prompt, conversation history, user messages and generated output. Most cloud-based service providers offer models which support very large context windows while limiting the maximum number of tokens that can be generated in a single response. For example, a model may support a context window of 1,000,000 tokens, allowing it to process very large conversations or documents, while limiting individual responses to 128,000 output tokens.

The context window and maximum token values are advisory. They are used when the service

provider does not return detailed limit information for a model. These values should be treated as general guidance and are not enforced by the library. The actual limits may vary depending on the provider, model version, deployment, enabled features and account limits. A value of zero for either ***nMaxTokens*** or ***nContextSize*** means those limits cannot be determined for the model. For locally hosted models or cloud-based aggregators, these limits may also be configurable for a specific model.

The metadata strings are provided for identification and display purposes. The ***szName*** member contains the canonical model name used in API requests, while ***szDisplayName*** provides a more user-friendly label when one is available. Other fields, such as ***szOwner***, ***szVersion***, ***szCreated***, and ***ftCreated***, are optional and may not be supplied by every provider.

Applications should set the ***dwSize*** member before passing this structure to a function. Unless otherwise documented, reserved members should be set to zero. When this structure is returned by the library, string members are stored directly in the fixed-length buffers contained within the structure, and may be read safely by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEnumModels, LlmGetModelInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_PROVIDER_INFO

The **LLM_PROVIDER_INFO** structure provides descriptive information and default connection settings for a language model provider.

```
typedef struct _LLM_PROVIDER_INFO
{
    DWORD   dwSize;
    DWORD   dwProviderId;
    DWORD   dwFlags;
    DWORD   dwReserved;
    LONG    nProtocol;
    LONG    nHostPort;
    TCHAR   szName[LLM_PROVIDER_NAMELEN];
    TCHAR   szHostName[LLM_PROVIDER_HOSTLEN];
    TCHAR   szDefaultModel[LLM_MODEL_NAMELEN];
} LLM_PROVIDER_INFO, *LPLLM_PROVIDER_INFO;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_PROVIDER_INFO)` before the structure is used.

dwProviderId

The provider identifier. This member should be one of the `LLM_PROVIDER` values. Refer to the providers page for a list of supported providers and their default models.

dwFlags

One or more provider capability flags. This member may contain a combination of `LLM_PROVIDER_FLAG` values.

dwReserved

A reserved value for future use. This member should always be zero.

nProtocol

The protocol used by the provider endpoint. Most cloud-based service providers require secure connections and applications should use `LLM_PROTOCOL_HTTPS`. For locally hosted models running on the same system or over a local network, `LLM_PROTOCOL_HTTP` can be used to establish a standard, non-secure connection.

nHostPort

The network port number associated with the provider endpoint. A value of zero indicates that the provider-specific default port should be used.

nHostPort

The network port number associated with the provider endpoint. A value of zero indicates that the provider-specific default port should be used.

szName

A null-terminated string which specifies a canonical provider name, such as `"OpenAI"`. This value may be used for identification or display in a user interface.

szHostName

A null-terminated string which specifies the default host name or IP address for the provider endpoint. This value may be an empty string for custom or application-defined providers.

szDefaultModel

A null-terminated string which specifies the default model name associated with the provider. This value may be an empty string if the provider requires the application to explicitly specify a model name.

Remarks

This structure is used to describe a language model provider, including its identity, endpoint information, and basic capabilities. It is typically populated by the library when enumerating known providers or retrieving information about a specific provider.

The ***dwProviderId*** member identifies the provider using one of the **`LLM_PROVIDER`** constants, while the ***dwFlags*** member indicates provider-level requirements or supported features, such as whether the provider requires TLS, requires authentication, supports enumerating models, or supports returning detailed model information. The ***nProtocol*** member identifies the default protocol used to communicate with the provider endpoint, such as HTTP or HTTPS.

The string members provide descriptive metadata about the provider. The ***szName*** member contains the canonical provider name. The ***szHostName*** member specifies the default network hostname associated with the provider, and ***szDefaultModel*** specifies the default model that may be used if the application does not explicitly choose one.

Not all providers supply the same level of detail. Some values may be empty or unspecified, particularly for custom, local, or application-defined providers. Applications should treat the contents of this structure as descriptive metadata and should not assume that every member will contain a meaningful value in all cases.

Applications should set the ***dwSize*** member before passing this structure to a function. Unless otherwise documented, reserved members should be set to zero. Because all string values are stored directly in the fixed-length buffers contained within the structure, the application may safely read or copy them as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmEnumProviders, LlmGetProviderInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_RESPONSE

The **LLM_RESPONSE** structure represents the response returned by a language model.

```
typedef struct _LLM_RESPONSE
{
    DWORD   dwSize;
    DWORD   dwRoleId;
    DWORD   dwFlags;
    DWORD   dwMessageId;
    DWORD   dwStatus;
    DWORD   dwElapsed;
    DWORD   dwReserved;
    TIME_T  tTimestamp;
    LONG    nOutputTokens;
    LONG    nContentLength;
    LPCTSTR lpszResponseId;
    LPCTSTR lpszModelName;
    LPCTSTR lpszStatusText;
    LPCTSTR lpszContent;
} LLM_RESPONSE, *LPLLM_RESPONSE;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_RESPONSE)` before the structure is used.

dwRoleId

The role associated with the response. This member will typically be set to `LLM_ROLE_ASSISTANT`, but may vary depending on the type of response returned.

dwFlags

Response flags. This member is reserved for future use and should typically be set to `LLM_RESPONSE_FLAG_NONE`.

dwMessageId

The identifier of the originating request message. This value corresponds to the *dwMessageId* member of the **LLM_MESSAGE** structure that initiated the request.

dwStatus

The response status code. This member should be one of the `LLM_RESPONSE` values, indicating how the request completed.

dwElapsed

The elapsed time, in milliseconds, required to process the request and generate the response.

dwReserved

A reserved value for future use. This member should always be zero.

tTimestamp

A 64-bit integer timestamp associated by the response. This member specifies when the response was received and the value is represented as the number of seconds from January 1, 1970 UTC.

nOutputTokens

The number of tokens generated in the response. This value is provided when token usage information is available.

nContentLength

The length of the response content, in characters, not including the terminating null character.

lpszResponseId

A null-terminated string which specifies a provider-defined identifier for the response. This value may be NULL.

lpszModelName

A null-terminated string which specifies the name of the model used to generate the response. This value may be NULL.

lpszStatusText

A null-terminated string which specifies a provider-specific status or diagnostic message. This value may be NULL.

lpszContent

A null-terminated string which specifies the response text returned by the model.

Remarks

The LLM_RESPONSE structure represents the result of a request submitted to a language model. It is typically returned by functions such as **LlmSendMessageEx** and provides both the generated response content and additional metadata describing how the request was processed.

This structure is logically paired with the **LLM_MESSAGE** structure. While the **LLM_MESSAGE** structure defines the input provided to the model, the **LLM_RESPONSE** structure contains the output generated by the model along with status and diagnostic information. The ***dwMessageId*** member may be used to associate the response with the originating request message.

The ***lpszContent*** member points to a null-terminated string buffer which contains the response from the model. The ***nContentLength*** member specifies how many characters are in the response, not including the terminating null character. Line feed and carriage return characters are converted to the standard Windows CRLF end-of-line sequence, ensuring consistent formatting across different providers and applications. This also helps ensure compatibility with Windows common controls and other user interface components.

The ANSI version of this structure uses UTF-8 encoded strings, while the Unicode version uses UTF-16 encoded strings. Response text returned by a provider may contain Unicode characters, including international text, typographical symbols, emoji, mathematical symbols, or other non-ASCII characters. Applications should ensure that any user interface controls, files, or other components used to display or process response text are capable of handling Unicode text correctly.

Applications which cannot process Unicode text in the response can provide a system prompt requesting that the model limit its output to ASCII characters. However, because model output is non-deterministic, applications should not assume that non-ASCII characters will never be returned.

The ***dwStatus*** member indicates how the request completed. A value of `LLM_RESPONSE_COMPLETE` indicates that the response was generated normally, while other values may indicate that the response was truncated due to token limits, cancelled, timed out, or failed due to an error. Applications should always check this value before processing the response content.

The ***dwElapsed*** member provides the total time required to process the request, which can be useful for performance monitoring or diagnostics. The ***nOutputTokens*** member, when available, indicates the number of tokens generated by the model, which may be useful for usage tracking or cost estimation.

The ***tTimestamp*** member is a standard 64-bit Unix time value which indicates when the response was

received from the service provider. It can be used interchangeably with functions in the C standard library such as **`gmtime`** or **`localtime`**. Because it is a 64-bit integer, it is not impacted by the Year 2038 problem. Timestamps can be converted into a local date and time using the **LlmConvertTimestamp** function.

The string members, such as *lpszContent*, *lpszResponseId*, *lpszModelName*, and *lpszStatusText*, are managed by the library. Applications should treat these pointers as read-only and must not attempt to modify the contents of the referenced strings. If an application needs to retain or modify this data, it should create its own copy.

Calling the **LlmClearHistory** or **LlmResetSession** functions will invalidate these string pointers. Applications must not reference these values after the session history has been cleared.

Unless otherwise documented, reserved members should be set to zero before calling a function which accepts this structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmGetFirstMessage, LlmGetMessage, LlmGetMessageById, LlmGetNextMessage, LlmSendMessageEx, LLM_MESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_SESSION

The **LLM_SESSION** structure contains extended configuration information used by the **LlmConnectEx** function to establish a session with a language model provider.

```
typedef struct _LLM_SESSION
{
    DWORD dwSize;
    DWORD dwProviderId;
    DWORD dwFlags;
    FLOAT fTemperature;
    FLOAT fTopP;
    LONG  nTopK;
    LONG  nTimeout;
    LONG  nIdleTime;
    LONG  nCacheTime;
    LONG  nMaxTokens;
    DWORD dwReasoningLevel;
    LPCTSTR lpszBaseUrl;
    LPCTSTR lpszModelName;
    LPCTSTR lpszApiKey;
    LPCTSTR lpszApiVersion;
    LPCTSTR lpszAppName;
    LPCTSTR lpszUserId;
    LPCTSTR lpszOrgId;
    LPCTSTR lpszProjectId;
    LPCTSTR lpszSystemPrompt;
} LLM_SESSION, *LPLLM_SESSION;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_SESSION)` before the structure is used. This value is validated by the library and is reserved for future structure versioning.

dwProviderId

The provider associated with the session. This member should be one of the `LLM_PROVIDER` values. If this value is `LLM_PROVIDER_NONE` the provider ID specified when calling the **LlmConnectEx** function will be used. Refer to the providers page for a list of supported providers and their default models.

dwFlags

Reserved flags for internal use. This member should be set to zero.

fTemperature

The temperature value used to control response randomness. A value of zero indicates that the provider or model default should be used. Unless you have a specific need to specify a temperature, it is recommended that you leave this value set to zero. Providers do not interpret temperature values identically and reasoning models may ignore the value entirely.

fTopP

The nucleus (top-p) sampling value. A value of zero disables top-p sampling and causes the default behavior to be used. It is recommended that you leave this value set to zero. Nucleus sampling is not used with most reasoning models and may be ignored. Although some providers still support setting a top-p value, most discourage using it with their current models.

nTopK

The top-k sampling value. A value of zero disables top-k sampling. It is recommended that you leave this value set to zero. Top-k sampling is not used with most reasoning models and in many cases the value is ignored. Although some providers still support setting a top-k value, most discourage using it with their current models.

nTimeout

The timeout period for requests, in seconds. A value of zero indicates that the default timeout should be used.

nIdleTime

The maximum idle time, in seconds, that an existing HTTP connection may be retained when the `LLM_OPTION_KEEPALIVE` connection option is enabled. If the elapsed idle time exceeds this value, the current connection will be closed and a new connection will be established for the next request. A value of zero indicates that the default idle timeout should be used.

nCacheTime

The length of time, in seconds, that the library may cache provider-specific information. This is primarily used when caching model lists returned by a provider. A value of zero indicates that the default cache behavior should be used.

nMaxTokens

The maximum number of tokens to request for generated output. A value of zero indicates that the provider or model default should be used.

dwReasoningLevel

Specifies the default reasoning level used for requests in this session. Higher levels may improve response quality for complex tasks but can increase latency and token usage.

lpszBaseUrl

A pointer to a null-terminated string which specifies the base URL for the provider endpoint. This member may be NULL to use the default endpoint for the selected provider.

lpszModelName

A pointer to a null-terminated string which specifies the model name to use for the session. For Microsoft Foundry models, this member typically specifies the deployment name. This member may be NULL if the provider defines a default model.

lpszApiKey

A pointer to a null-terminated string which specifies the API key or authentication token used to access the provider. This member may be NULL if authentication is not required or is provided by another mechanism.

lpszApiVersion

A pointer to a null-terminated string which specifies the API version associated with the provider. This member may be NULL if no explicit version is required.

lpszAppName

A pointer to a null-terminated string which specifies an optional application name associated with the session. When provided, this value may be included in the user-agent string sent with requests. This value may be sanitized by the library before use and may be NULL.

lpszUserId

A pointer to a null-terminated string which specifies an optional user identifier associated with the session. This member may be NULL.

lpszOrgId

A pointer to a null-terminated string which specifies an optional organization identifier associated with the session. This member may be NULL.

lpszProjectId

A pointer to a null-terminated string which specifies an optional project identifier associated with the session. This member may be NULL.

lpszSystemPrompt

A pointer to a null-terminated string which specifies an optional system prompt used to define the initial behavior of the model for the session. This member may be NULL.

Remarks

This structure is used with the **LlmConnectEx** function to provide extended configuration information when creating a session. It allows an application to customize connection behavior, sampling parameters, model selection, authentication, and other optional settings without requiring a large number of function parameters. Applications that only need to specify the provider, base URL, model name, and API key may use the simpler **LlmConnect** function instead.

The *dwProviderId* member should normally match the provider specified in the **LlmConnectEx** function call. A value of `LLM_PROVIDER_NONE` may be used when the application relies on the function parameter to select the provider. If conflicting provider values are specified, the function will fail.

In most cases, applications only need to initialize *dwSize*, *dwProviderId*, and the specific members they wish to override. All unused numeric members should be set to zero, and all unused string members should be set to NULL. The library will apply provider-specific defaults when optional values are not specified.

The *lpszApiKey* member is a string value that specifies the API key used to authenticate with the provider. The API key may be provided using a literal string, an environment variable, or a value stored in the Windows Credential Manager. Using an environment variable or secure credential storage is recommended for production applications to avoid exposing sensitive information in source code or application binaries. To specify an environment variable, surround the name in percent (%) symbols, such as `"%OPENAI_API_KEY%"`. To specify a Windows Credential, surround it in braces, such as `"{MyAppName/OpenAI}".`

For production applications, it is recommended that you avoid using literal API key strings and instead use an environment variable or a value stored in the Windows Credential Manager. If you must provide the API key as a literal string, you should take additional steps to protect it, such as storing the key in an encrypted form and limiting access to authorized users or processes. Avoid embedding API keys directly in application source code, configuration files, installers or other content that could be exposed through version control systems, logs or public repositories.

The *dwReasoningLevel* member provides a provider-neutral hint that controls how much internal reasoning a model should use when generating a response. Higher levels may improve the quality of results for more complex tasks, but can also increase response time and token usage. Not all models support reasoning controls, and the exact behavior is provider-specific. In most cases, it is recommended that you set this value to LLM_REASONING_DEFAULT, allowing the provider or model to determine the appropriate level automatically.

The sampling-related members *fTemperature*, *fTopP*, and *nTopK* control how responses are generated. In most applications, adjusting the temperature alone is sufficient, while Top-P and Top-K are intended for more advanced tuning scenarios. If these values are not specified, the provider or model defaults are used. Not all providers or models support sampling parameters such as temperature, top-p or top-k. The library will only include those values when they are supported by the selected provider and model. For models that do not support these parameters, the session values are

ignored.

The connection-related members *lpszBaseUrl*, *nTimeout*, and *nIdleTime* control how requests are sent and how long an existing HTTP connection may be reused. When the `LLM_OPTION_KEEPALIVE` option is enabled, connections may be reused across multiple requests, subject to the idle timeout period.

The *nCacheTime* member controls how long the library may cache provider-specific information, such as model lists. This helps reduce unnecessary network requests when applications repeatedly query provider capabilities.

The *lpszAppName* member specifies the application name associated with requests sent by the control. This value is included as part of the default user-agent string and may be used for diagnostic and identification purposes by the provider. A recommended format is to specify both the application name and version number using a convention similar to standard user-agent strings, for example: `"MyApplication/1.0"`.

The *lpszUserId* member is a user identifier associated with requests sent to the current provider. Some providers allow applications to specify a user identifier as part of the request payload for diagnostics, analytics, rate limiting, or request tracking. Some providers may ignore the value or use it only for internal monitoring and logging purposes. Applications should avoid including sensitive or personally identifiable information in the user identifier value unless required by the provider.

The *lpszOrgId* member specifies an optional organization, tenant or account identifier used by some providers when processing requests. This is primarily intended for enterprise or multi-tenant environments where a provider account may be associated with multiple organizations or projects. Most applications will not need to set this value unless specifically required by the provider. If the current provider does not support organization identifiers, the value is ignored.

The *lpszProjectId* member specifies an optional project identifier used by some providers when processing requests. This is primarily intended for providers which support project-based authentication, billing or resource management. Most applications will not need to set this value unless specifically required by the provider or account configuration. If the current provider does not support project identifiers, the value is ignored.

The *lpszSystemPrompt* member may be used to define a session-level system prompt that establishes the behavior of the model. A system prompt provides instructions that are included with each request sent to the model. It can be used to define the expected role, tone, response style, formatting rules, or other application-specific guidance that should apply to the session. For example, an application might use a system prompt to request short responses, avoid Markdown formatting, return structured text, or answer only within the context of a specific feature or workflow.

The string members in this structure are supplied by the application and are treated as input values when the session is created. Applications should ensure that any referenced strings remain valid until a connection is established. The library makes an internal copy of these values, so the application does not need to maintain them after the connection completes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmConnectEx, LlmCreateSession, LlmCreateSessionEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

LLM_TOKEN_USAGE

The **LLM_TOKEN_USAGE** structure contains information about token usage for a client session.

```
typedef struct _LLM_TOKEN_USAGE
{
    DWORD   dwSize;
    DWORD   dwProviderId;
    DWORD   dwReserved;
    LONG    nMessageCount;
    LONG    nInputTokens;
    LONG    nOutputTokens;
    LONG    nContextUsed;
    LONG    nContextSize;
    TCHAR   szModelName[LLM_MODEL_NAMELEN];
} LLM_TOKEN_USAGE, *LPLLM_TOKEN_USAGE;
```

Members

dwSize

The size of this structure, in bytes. This member must be set to `sizeof(LLM_TOKEN_USAGE)` before the structure is used.

dwProviderId

The provider identifier. This member should be one of the `LLM_PROVIDER` values. Refer to the providers page for a list of supported providers and their default models.

dwReserved

A reserved value for future use. This member will always be zero.

nMessageCount

The number of messages currently stored in the session history.

nInputTokens

The number of input tokens associated with system and user messages in the current session.

nOutputTokens

The number of output tokens generated in model responses for the current session.

nContextUsed

The total number of tokens currently used in the active chat context, including input messages and message history.

nContextSize

The maximum number of tokens supported by the current model for the active context. A value of zero indicates that this limit is not available.

szModelName

A null-terminated string which specifies the name of the model associated with the current session. This value may be empty if no model name is available.

Remarks

This structure is used to return token usage information for an active client session. It provides a summary of how many messages are currently in the session history, how many tokens have been used for input and output, and how much of the available context window is currently in use.

The ***nInputTokens*** member reflects tokens associated with user and system input, while

nOutputTokens reflects tokens generated by the model in its responses. The *nContextUsed* member represents the total token usage currently contributing to the active chat context, including prior message history that may be sent with each request.

The *nContextSize* member indicates the maximum context size supported by the current model, when that information is available. Applications may compare *nContextUsed* with *nContextSize* to estimate how much of the available context window has been consumed.

This structure is intended for informational and diagnostic use. Token counts are provider-dependent and may represent estimates rather than exact values, depending on the model or service in use. Applications should not rely on these values to enforce strict limits, as the provider ultimately determines whether a request exceeds its supported context or token constraints.

Applications should set the *dwSize* member before passing this structure to a function. Unless otherwise documented, reserved members should be set to zero. String values are stored directly in the fixed-length buffers contained within the structure and may be read safely by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

LlmSendMessage, LlmSendMessageEx, LlmGetTokenUsage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Constants

Constant	Value	Description
Ports		
LLM_PORT_DEFAULT	0	Use the provider-specific default port.
LLM_PORT_SECURE	443	Default port for secure HTTPS connections.
Protocols		
LLM_PROTOCOL_DEFAULT	0	Use the provider-specific default protocol.
LLM_PROTOCOL_HTTP	1	Use the HTTP protocol.
LLM_PROTOCOL_HTTPS	2	Use the HTTPS protocol.
LLM_PROTOCOL_UNKNOWN	-1	Unknown or invalid protocol.
Options		
LLM_OPTION_NONE	0	No options specified.
LLM_OPTION_NOCACHE	0x000001	Disable provider and model caching for the session.
LLM_OPTION_KEEPALIVE	0x000002	Enable persistent (keep-alive) HTTP connections.
LLM_OPTION_PROXY	0x000008	Use a configured proxy server for the connection.
LLM_OPTION_NOUSERAGENT	0x000020	Do not send a User-Agent header with the request.
LLM_OPTION_SECURE	0x001000	Force use of a secure (HTTPS) connection.
LLM_OPTION_DEFAULT	LLM_OPTION_NONE	Default option set (no flags enabled).
Providers		
LLM_PROVIDER_NONE	0	No provider specified.
LLM_PROVIDER_LOCAL	1	Local inference provider (e.g., LM Studio or Ollama).
LLM_PROVIDER_OPENAI	2	OpenAI provider using the ChatGPT models.

LLM_PROVIDER_ANTHROPIC	3	Anthropic provider using the Claude models.
LLM_PROVIDER_GOOGLE	4	Google AI provider using the Gemini models.
LLM_PROVIDER_MICROSOFT	5	Microsoft Foundry provider using OpenAI-compatible models.
LLM_PROVIDER_MISTRAL	6	Mistral AI provider using the Mistral models.
LLM_PROVIDER_UNKNOWN	((DWORD)-1)	Unknown or unrecognized service provider.
LLM_PROVIDER_DEFAULT	LLM_PROVIDER_LOCAL	Default service provider.
Reasoning		
LLM_REASONING_NONE	0	No reasoning, even if it is available.
LLM_REASONING_DEFAULT	1	Provider or model default.
LLM_REASONING_LOW	2	Prefer faster responses; reduces usage costs.
LLM_REASONING_MEDIUM	3	Balanced reasoning.
LLM_REASONING_HIGH	4	Prefer quality responses; increases usage costs.
LLM_REASONING_MAXIMUM	5	Maximum reasoning quality.
Temperature		
LLM_TEMPERATURE_MINIMUM	0.0	Deterministic output with minimal randomness.
LLM_TEMPERATURE_CONSERVATIVE	0.3	Low variability, more predictable responses.
LLM_TEMPERATURE_BALANCED	0.7	Balanced creativity and coherence.
LLM_TEMPERATURE_DEFAULT	1.0	Standard default sampling behavior.
LLM_TEMPERATURE_CREATIVE	1.2	Increased creativity and variation.
LLM_TEMPERATURE_MAXIMUM	2.0	Maximum randomness with highly varied output.
Top-P Sampling		

LLM_TOP_P_DISABLED	0.0	Nucleus sampling disabled.
LLM_TOP_P_MINIMUM	0.1	Very restrictive token selection.
LLM_TOP_P_CONSERVATIVE	0.8	Conservative token selection.
LLM_TOP_P_BALANCED	0.9	Balanced nucleus sampling.
LLM_TOP_P_CREATIVE	0.95	More diverse token selection.
LLM_TOP_P_MAXIMUM	1.0	Full probability distribution with no filtering.
LLM_TOP_P_DEFAULT	LLM_TOP_P_DISABLED	Default disables nucleus sampling.
Top-K Sampling		
LLM_TOP_K_DISABLED	0	Top-K sampling disabled.
LLM_TOP_K_MINIMUM	0	Equivalent to disabled sampling.
LLM_TOP_K_CONSERVATIVE	10	Restrict to top 10 tokens.
LLM_TOP_K_BALANCED	40	Balanced token selection range.
LLM_TOP_K_CREATIVE	60	More diverse token selection.
LLM_TOP_K_MAXIMUM	100	Wide token selection range.
LLM_TOP_K_DEFAULT	LLM_TOP_K_DISABLED	Default disables Top-K sampling.
Frequency		
LLM_FREQUENCY_DISABLED	0.0	No frequency penalty applied.
LLM_FREQUENCY_MINIMUM	0.0	Minimum frequency penalty.
LLM_FREQUENCY_MAXIMUM	2.0	Maximum frequency penalty.
LLM_FREQUENCY_DEFAULT	LLM_FREQUENCY_DISABLED	Default disables frequency penalty.
Presence		
LLM_PRESENCE_DISABLED	0.0	No presence penalty

		applied.
LLM_PRESENCE_MINIMUM	0.0	Minimum presence penalty.
LLM_PRESENCE_MAXIMUM	2.0	Maximum presence penalty.
LLM_PRESENCE_DEFAULT	LLM_PRESENCE_DISABLED	Default disables presence penalty.
Provider Flags		
LLM_PROVIDER_FLAG_NONE	0	No provider requirements specified.
LLM_PROVIDER_FLAG_REQUIRE_TLS	0x0001	Requires a secure TLS (HTTPS) connection.
LLM_PROVIDER_FLAG_REQUIRE_AUTH	0x0002	Requires authentication (e.g., API key or token).
LLM_PROVIDER_FLAG_API_VERSION	0x0004	Requires an explicit API version parameter.
LLM_PROVIDER_FLAG_LOCAL	0x0008	Locally hosted provider (LM Studio, Ollama, etc.)
LLM_PROVIDER_FLAG_MODEL_LIST	0x0010	Supports enumerating available models.
LLM_PROVIDER_FLAG_MODEL_INFO	0x0020	Supports retrieving detailed model information.
Model Options		
LLM_MODEL_OPTION_DEFAULT	0x0000	default list of chat models for a provider.
LLM_MODEL_OPTION_PREVIEW	0x0001	Include preview or experimental models.
LLM_MODEL_OPTION_STREAMING	0x0002	Include models that support streaming responses.
LLM_MODEL_OPTION_STRUCTURED	0x0004	Include models that support structured outputs.
LLM_MODEL_OPTION_REASONING	0x0008	Include reasoning-oriented models.
LLM_MODEL_OPTION_VISION	0x0010	Include models that support image input.
LLM_MODEL_OPTION_AUDIO	0x0200	Include models that support audio output (text-to-speech).

LLM_MODEL_OPTION_IMAGE	0x0400	Include models that support image generation or input.
LLM_MODEL_OPTION_VIDEO	0x0800	Include models that support video generation.
LLM_MODEL_OPTION_TRANSCRIPTION	0x1000	Include models that support speech-to-text transcription.
LLM_MODEL_OPTION_DEPRECATED	0x8000	Include deprecated or retired models.
LLM_MODEL_OPTION_ALL_MODELS	0xFFFF	Include all available model types.
Model Flags		
LLM_MODEL_FLAG_NONE	0x0000	No model capabilities defined.
LLM_MODEL_FLAG_CHAT	0x0001	Supports conversational text generation.
LLM_MODEL_FLAG_SYSTEM_ROLE	0x0002	Accepts a system role message in the input.
LLM_MODEL_FLAG_STREAMING	0x0004	Supports streaming responses.
LLM_MODEL_FLAG_STRUCTURED	0x0008	Supports structured output (e.g., JSON or schema).
LLM_MODEL_FLAG_VISION	0x0010	Supports image inputs (multimodal).
LLM_MODEL_FLAG_TOOLS	0x0020	Supports tool or function calling.
LLM_MODEL_FLAG_REASONING	0x0040	Supports reasoning-oriented generation.
LLM_MODEL_FLAG_RESPONSES	0x0080	Supports the Responses API (unified endpoint).
LLM_MODEL_FLAG_FUNCTIONS	0x0100	Supports OpenAI-style function calling with schemas.
LLM_MODEL_FLAG_AUDIO	0x0200	Supports audio output (text-to-speech)
LLM_MODEL_FLAG_IMAGE	0x0400	Supports image generation.
LLM_MODEL_FLAG_VIDEO	0x0800	Supports video generation.

LLM_MODEL_FLAG_TRANSCRIPTION	0x1000	Supports speech-to-text transcription.
LLM_MODEL_FLAG_INSTRUCTIONS	0x2000	Supports a dedicated instruction field outside message lists.
LLM_MODEL_FLAG_MAX_COMPLETION_TOKENS	0x4000	Uses `max_completion_tokens` instead of `max_tokens` in requests.
LLM_MODEL_FLAG_INFERRED	0x80000	Capabilities are inferred rather than explicitly defined.
Model Status		
LLM_MODEL_STATUS_NONE	0x0000	No status assigned.
LLM_MODEL_STATUS_AVAILABLE	0x0001	Model is actively available.
LLM_MODEL_STATUS_PREVIEW	0x0002	Model is in preview or experimental stage.
LLM_MODEL_STATUS_DEPRECATED	0x0004	Model is deprecated or legacy.
Limits		
LLM_MAX_MESSAGE_LENGTH	65536	Maximum length of a single input message (64 KB)
LLM_MAX_RESPONSE_LENGTH	262144	Maximum expected response length (256 KB)
LLM_MAX_SYSTEM_PROMPT_LENGTH	32768	Maximum length of the system prompt (32 KB)
LLM_MAX_CHAT_HISTORY	1000	Maximum number of messages stored per session.
LLM_MAX_TOKENS	200000	Maximum token limit per request.
Roles		
LLM_ROLE_UNKNOWN	0	Unknown or invalid role.
LLM_ROLE_USER	1	User input message.
LLM_ROLE_ASSISTANT	2	Assistant response message.
LLM_ROLE_SYSTEM	3	System-level instruction message.

LLM_ROLE_FUNCTION	4	Function call message (legacy)
LLM_ROLE_TOOL	5	Tool-generated message.
LLM_ROLE_TOOLUSER	6	User input directed to a tool.
LLM_ROLE_DEFAULT	LLM_ROLE_USER	Default role is user input.
Responses		
LLM_RESPONSE_UNKNOWN	0	No stop reason specified or unrecognized response state.
LLM_RESPONSE_COMPLETE	1	Response completed normally.
LLM_RESPONSE_LIMITED	2	Stopped due to token or context limits.
LLM_RESPONSE_ABORTED	3	Request was cancelled or interrupted.
LLM_RESPONSE_ERROR	4	An error occurred during processing.
LLM_RESPONSE_TIMEOUT	5	Request timed out.
Session		
LLM_SESSION_FLAG_NONE	0	No session flags specified.
LLM_SESSION_FLAG_DEFAULT	LLM_SESSION_FLAG_NONE	Default session behavior.
Cache		
LLM_INVALID_CACHE_TIME	-1	Indicates an invalid or unspecified cache duration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Large Language Model Providers

One of the first decisions you will need to make when using the SocketTools Large Language Model API is choosing which service provider to use. The library is designed to support a variety of providers and models, and the provider type is specified using one of following provider constants. The current version supports the following providers:

Constant	Value	Default Model	Description
LLM_PROVIDER_LOCAL	1	None	Locally hosted models using servers like LM Studio or Ollama. This is ideal for development, testing, offline use, or avoiding usage costs. A wide range of open-source models are available; for best results, start with smaller instruction-tuned models such as **qwen2.5-7b-instruct**, **llama-3.2-3b-instruct**, or **mistral-7b-instruct**.
LLM_PROVIDER_OPENAI	2	**gpt-5.4-nano**	OpenAI's public API (ChatGPT) is a recommended starting point for most developers due to ease of use, strong general performance, and comprehensive documentation. For initial development and testing, lightweight models such as the GPT-5 mini series provide fast response times and lower usage costs.
LLM_PROVIDER_ANTHROPIC	3	**claude-haiku-4-5**	Anthropic's Claude models are well suited for structured reasoning, conversational clarity, and safety-focused applications. For initial use, the Haiku models offer lower cost and faster response times while maintaining strong performance.
LLM_PROVIDER_GOOGLE	4	**gemini-2.5-flash**	Google's Gemini models are a good choice for multimodal applications and integration with the Google AI ecosystem. The Flash models are recommended for getting started due to their lower latency and reduced token costs.
LLM_PROVIDER_MICROSOFT	5	None	Microsoft Foundry OpenAI-compatible models are typically used in enterprise environments requiring Azure integration and regional deployment control. This option requires provisioning a model deployment within Foundry and specifying the deployment name and API version when connecting.
LLM_PROVIDER_MISTRAL	6	**mistral-small-3.2**	Mistral AI models provide efficient and cost-effective options for general-purpose and

			reasoning tasks. Smaller models are recommended when getting started due to their performance and lower cost.

Choosing a Model

Although SocketTools provides a default model for most providers, the model names and availability will change over time. The providers which do not have a default model assigned to them will always require a specific model name when establishing the connection. Some providers, notably Microsoft Foundry and locally hosted services, will explicitly require you to provide a model name to establish a connection.

It is important to note that the library is not limited only to those default models listed above. Developers are encouraged to enumerate available models and explicitly select a model that meets their requirements. The library provides functions designed to return information about all of the compatible models supported by a provider. For more information refer to the LlmEnumModels function.

Most language model providers offer multiple models with different capabilities, performance characteristics, and pricing. Rather than selecting a specific model name immediately, it is often more useful to choose a general category of model based on your application's needs.

Common model categories include:

- **Lightweight**. Optimized for speed and low cost. These are ideal for simple tasks such as formatting, summarization, classification, or user interface interactions where responsiveness is important. These are the default models assigned to supported providers. In most cases, we recommend you start with a lightweight model until you determine your application could benefit from a more capable model type.
- **Balanced**. General purpose models that provide good performance across a wide range of tasks. These are typically the best default choice for most applications which need to balance between performance and capability. Examples are **`gpt-5-mini`**, **`claude-sonnet-4-6`**, **`gemini-3.1`** and **`mistral-medium`** models.
- **Reasoning**. Designed for more complex tasks such as multi-step problem solving, analysis, and code generation. These models may produce more detailed responses but typically use more tokens and have higher latency and increased costs. Examples are **`gpt-5`**, **`claude-opus-4-7`**, **`gemini-3.1-pro`** and **`mistral-large`** models.
- **Multimodal**. Support input or output beyond plain text, such as images, audio, or structured data. These are useful when working with media content or advanced integrations.

In most cases, developers should begin with a lightweight or balanced model and only move to more advanced models when additional capability is required. The exact model names and capabilities will vary between providers, but these categories provide a useful framework for selecting an appropriate model. Applications should also avoid relying on strict token limits or attempting to precisely manage context usage. Providers ultimately determine request limits, and applications should be prepared to handle responses that are truncated or rejected when limits are exceeded.

API Keys and Authentication

Most language model providers require an API key (or authentication token) to access their services. An API key is a unique identifier associated with your account that is included with each request to authenticate the client and track usage.

When using the library, the API key is typically provided when establishing a session using the **LlmConnect** or **LlmConnectEx** function. The library will include this key in requests to the selected provider using the appropriate authentication mechanism.

Because API keys grant access to external services and may incur usage charges, they should be handled carefully and not exposed unnecessarily. The library supports several functions for providing API keys, allowing applications to balance convenience and security.

Providing an API Key

The library supports the following functions for supplying an API key:

- **Literal string**
  The API key can be provided directly as a string when creating a session. This is the simplest approach and is often used for testing or development, but it is not recommended for production applications because the key may be exposed in application binaries or source code.
- **Environment variable**
  The API key can be stored in an environment variable and retrieved at runtime. This approach avoids embedding the key directly in the application and is commonly used in development environments, scripts, or deployment pipelines.
- **Windows Credential Manager**
  The API key can be stored securely using the Windows Credential Manager and retrieved by the application at runtime. This is the recommended approach for Windows applications because it provides secure storage and helps prevent accidental exposure of sensitive credentials.

Best Practices

- Do not embed API keys directly in source code for production applications.
- Use secure storage mechanisms, such as the Windows Credential Manager, whenever possible.
- Limit access to API keys and avoid sharing them between applications or users.
- Review provider usage policies and configure usage limits or alerts where available.

Each provider has its own process for creating and managing API keys. Refer to the provider documentation for details on generating keys, rotating credentials, and managing access.

Application Design

When designing an application that uses a language model provider, developers generally have two approaches for managing API keys:

- **Application-managed API key**
  The developer provides an API key that is used by the application for all requests. This approach is simple to implement and provides a consistent user experience, but the developer is responsible for all usage and any associated costs.
- **User-provided API key (Bring Your Own Key)**
  The application allows the end user to supply their own API key, typically through a configuration dialog or settings interface. In this case, the user is responsible for usage and costs, and the application acts as a client using the user's credentials.

Each approach has trade-offs. Using an application-managed key simplifies the user experience but may require safeguards such as rate limiting or usage monitoring. Allowing users to provide their own

key can reduce operational costs and provide greater flexibility, but requires additional user interface considerations and clear instructions for obtaining and configuring API access.

Applications intended for a broad audience often prefer a built-in API key for ease of use, while developer-focused tools frequently support user-provided keys for flexibility.

Provider Overview

The following sections provide more detailed information about the supported providers. One of the advantages of the library is that applications are not limited to a single language model service. Different providers may offer advantages related to pricing, performance, regional availability, deployment flexibility, or model capabilities.

Supporting multiple providers can also provide greater flexibility for end users. Some users or organizations may already have existing accounts, billing arrangements, or infrastructure associated with a specific provider such as OpenAI, Google, Microsoft Foundry, or a locally hosted inference service. Applications can allow users to select the provider and model which best meets their requirements without requiring significant changes to application logic.

LLM_PROVIDER_LOCAL

This provider type is used for locally hosted language model inference. This is particularly useful for development, testing, and experimentation without relying on a cloud-based provider. Local hosting allows developers to evaluate model behavior, test integrations, and work offline or within a private environment. This requires you to install and configure an application on your local system or network which will provide the inference service. For the Windows platform, we recommend one of the following applications, which are free to install and use:

LM Studio

LM Studio provides a full-featured desktop environment for running and testing language models locally. It includes model management, a built-in server, and support for OpenAI-compatible APIs, making it a strong choice for development and evaluation.

- LM Studio
- LM Studio Documentation

Ollama

Ollama is a lightweight local model runtime that makes it easy to download and run models on a local system. It provides a simple API interface and is well suited for basic testing and self-hosted deployments.

- Ollama
- Ollama Documentation

LLM_PROVIDER_OPENAI

The OpenAI provider connects to OpenAI's public API and is generally the recommended starting point for most developers. It offers a wide range of well-documented models, consistent API behavior, and strong general-purpose performance. Developers who are new to using large language models or want a reliable default should typically begin here.

To get started, developers create an API key, select a model name, and connect to the default OpenAI endpoint. The API is widely supported and serves as the baseline for compatibility with many other providers.

- OpenAI Platform

- OpenAI Documentation
- Manage API Keys

LLM_PROVIDER_ANTHROPIC

The Anthropic provider connects to Anthropic's Claude API, which focuses on conversational models designed for safety and structured reasoning. Developers must create an API key and select a supported Claude model. The API is similar in concept to OpenAI but uses its own endpoints and request formats.

- Anthropic
- Claude Documentation

LLM_PROVIDER_GOOGLE

The Google provider connects to Google's Gemini API, which provides access to multimodal models and integrates with the broader Google AI ecosystem. Developers typically obtain an API key through Google AI Studio and use it to authenticate requests to the Gemini service.

- Google AI
- Google Documentation

LLM_PROVIDER_MICROSOFT

The Microsoft provider connects to a Microsoft Foundary OpenAI-compatible deployment. This option is often used in enterprise environments where Azure integration, regional deployment, or Microsoft-managed infrastructure is required. Unlike the public OpenAI API, Microsoft requires a resource-specific endpoint, a deployment name instead of a direct model name, and an explicit API version. Authentication is typically performed using an API key.

- Microsoft Foundry
- Azure OpenAI Reference

LLM_PROVIDER_MISTRAL

The Mistral provider connects to Mistral AI's hosted API service. Mistral models are designed to provide strong general-purpose and reasoning performance while maintaining efficient resource usage and competitive pricing. Developers who want a balance between capability, response speed, and lower operational cost may find these models to be a good alternative to larger commercial providers. Some organizations may prefer Mistral AI due to regional deployment considerations or compatibility with European-based infrastructure and services.

- Mistral AI
- Mistral Documentation
- Mistral Console

See Also

- Functions
- Constants
- Data Structures

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Message History File Format

The Large Language Model client API uses a JSON-formatted message history file to support exporting and importing conversation state between application sessions. This format is used by the **LlmExportHistory** and **LlmImportHistory** functions and is intended to provide a simple, human-readable representation of the conversation history associated with a client session.

A message history file contains both the messages sent by the client application and the responses generated by the model. Together, these entries represent the conversational context for the session. The file does not contain information about the client session, aside from the provider type and model name. It does not contain user names, authorization tokens or any information related to the service account.

The history file may contain sensitive or private information, including user prompts, model responses or application-specific data. Applications should take reasonable precautions to protect these files, particularly if they are stored, transmitted or made available for users to download.

The following example shows a simplified message history file containing a single message turn:

```
{
  "version": 1,
  "provider": 1,
  "created": 1715000000,
  "messages": [
    {
      "message": {
        "role": 1,
        "reasoning": 0,
        "flags": 0,
        "id": 1,
        "timestamp": 1715000001,
        "maxTokens": 0,
        "inputTokens": 12,
        "content": "Hello"
      },
      "response": {
        "role": 2,
        "flags": 0,
        "messageId": 1,
        "status": 1,
        "elapsed": 812,
        "timestamp": 1715000002,
        "outputTokens": 24,
        "model": "gpt-5.4-nano",
        "content": "Hi there."
      }
    }
  ],
  "messageCount": 1,
  "model": "gpt-5.4-nano"
}
```

File Structure

A message history file consists of a root JSON object which contains session metadata and an array of message turns.

version

The format version number used by the exported history file.

provider

The numeric provider identifier associated with the session.

created

A Unix timestamp value which indicates when the history file was created.

messages

An array containing the exported message turns.

messageCount

The number of message turns stored in the history file.

model

The default model name associated with the exported session.

Message Turns

Each entry in the **`messages`** array represents a complete message turn consisting of two objects:

- A **`message`** object which contains the client input
- A **`response`** object which contains the model output

The fields within these objects closely correspond to members of the **`LLM_MESSAGE`** and **`LLM_RESPONSE`** structures used internally by the API.

Applications should not depend on undocumented fields or the exact formatting of exported history files. Future versions of the library may add additional fields or metadata while maintaining compatibility with previous versions where possible.

Remarks

The message history format is intended primarily for exporting, importing and persisting conversation state between application sessions. Although the file format is human-readable JSON text, applications should avoid modifying exported history files unless they fully understand the relationship between the stored values.

Changing values such as the provider identifier, model name, message role or status values may produce undefined or unexpected results when the history is imported. Applications should not modify role identifiers or assume that numeric values used internally by the API will remain unchanged in future versions.

Modifying the message or response content may also invalidate related metadata such as token usage values. For example, changing the **`content`** field may result in incorrect **`inputTokens`** or **`outputTokens`** values being associated with the imported conversation history.

See Also

LlmExportHistory, LlmImportHistory, LLM_MESSAGE, LLM_RESPONSE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Mail Message Library

Compose and parse standard MIME formatted email messages.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSMSGV12.DLL
Version	12.0.3024.1286
LibID	123A728A-D6CD-435F-A76D-0BA3C07DDFC1
Import Library	CSMSGV12.LIB
Dependencies	None
Standards	RFC 822, RFC 2045, RFC 2046, RFC 2047, RFC 2048

Overview

The Mail Message library provides an interface for composing and processing email messages and newsgroup articles which are structured according to the Multipurpose Internet Mail Extensions (MIME) standard. Using this library, an application can easily create complex messages which include multiple alternative content types, such as plain text and styled HTML text, file attachments and customized headers.

It is not required that the developer understand the complex MIME standard; a single function call can be used to create multipart message, complete with a styled HTML text body and support for international character sets. The Mail Message library can be easily integrated with the other mail related protocol libraries, making it extremely easy to create and process MIME formatted messages.

The library also includes an interface for managing a local message storage file that can be used to store and retrieve multiple messages. Functions are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Mail Message Functions

Function	Description
MimeAddMessageHeaders	Add one or more header values to the current message
MimeAppendMessageText	Append text to the body of the current message part
MimeAttachData	Attach the contents of a buffer to the current message
MimeAttachFile	Attach a file to the current message
MimeAttachImage	Attach an inline image to the current message
MimeCheckAuthenticationResults	Check the authentication status for the current message
MimeClearMessageText	Clear the body of the current message part
MimeCloseMessage	Close the message handle and release the resources allocated for the message
MimeCloseMessageStore	Close the current message storage file
MimeCompareMessageText	Compare text in the body of the current message part
MimeComposeMessage	Compose a new message using the specified parameters
MimeCopyMessageStore	Duplicate the contents of the current message store in a new file
MimeCreateMessage	Create a new empty message
MimeCreateMessageEx	Create a new message from the contents of a file or a message in memory
MimeCreateMessagePart	Create a new message part for the current message
MimeCreateMessagePartEx	Create a new message part for the current message
MimeDecodeText	Decode a base64 or quoted-printable encoded string
MimeDecodeTextEx	Decode a base64 or quoted-printable encoded string using a specified character set
MimeDeleteMessageHeader	Delete the specified header field from the message
MimeDeleteMessagePart	Delete the current message part
MimeDeleteStoredMessage	Remove the current message from the message store
MimeEncodeText	Encode a string using base64 or quoted-printable encoding
MimeEncodeTextEx	Encode a string using base64 or quoted-printable encoding using a specified character set
MimeEnumAttachments	Enumerate all file attachments in the current message
MimeEnumMessageHeaders	Enumerate all header fields in the current message part
MimeEnumMessageRecipients	Enumerate addresses of all message recipients
MimeExportMessage	Export the current message to a text file
MimeExportMessageEx	Export the current message to a file, clipboard or memory buffer
MimeExtractAllFiles	Extract all file attachments in the message and store them in the specified directory
MimeExtractFile	Extract the file attachment from the current message part
MimeExtractFileEx	Extract a file attachment from the message with additional options
MimeFilterMessageText	Filter the contents of a HTML or plain text message
MimeFindAttachment	Search for a file attachment in the current message
MimeFindMessageText	Search for a regular expression or string literal in the current message text
MimeFindMessageTextEx	Saerch for a regular expression or string literal and return the offset and length into the message contents
MimeFindStoredMessage	Search for a message in the current message store
MimeFormatDate	Return a standard date and time string using a 32-bit time value
MimeFormatDate64	Return a standard date and time string using a 64-bit time value
MimeGetAllHeaders	Return the complete RFC 822 header values in a string buffer
MimeGetAllRecipients	Return a comma-separated list of recipient addresses in a string buffer
MimeGetAttachedFileName	Return the name of the file attachment for the current part

MimeGetAuthenticationResults	Return information about the authentication status of the message
MimeGetContentDigest	Return encoded digest of message's content
MimeGetContentLength	Return the length of the current message part content
MimeGetContentType	Return the content type identifier for the current message part
MimeGetContentTypeEx	Return information about the content type for the current message part
MimeGetErrorString	Return a description for the specified error code
MimeGetExportOptions	Return a bitmask that describes current message export options
MimeGetFileContentType	Return the content type for a specified file
MimeGetFirstMessageHeader	Return the first header field and value in the current message part
MimeGetLastError	Return the last error code
MimeGetMessageBoundary	Return the multipart message boundary string
MimeGetMessageDate	Return the date and time from the message header
MimeGetMessageDateEx	Return a localized message date and time to a string buffer
MimeGetMessageHeader	Return the value of a specified header from the message
MimeGetMessageHeaderEx	Copy the value of specified header to a string buffer
MimeGetMessageLinks	Returns the hyperlinks in the current message part
MimeGetMessagePart	Return the current message part index
MimeGetMessagePartCount	Return the total number of message parts
MimeGetMessageSender	Return the email address of the message sender
MimeGetMessageSize	Return the size of the complete message in bytes
MimeGetMessageText	Return the text of the current message part
MimeGetMessageVersion	Return the MIME version from the message header
MimeGetNextMessageHeader	Return the next header field and value in the current message part
MimeGetStoredMessage	Retrieve a message from the current message store
MimeGetStoredMessageCount	Return the number of messages in a message store
MimeImportMessage	Import a message from the specified text file
MimeImportMessageEx	Import a message from a file, clipboard or memory buffer
MimeInitialize	Initialize the library for use by the client
MimeIsContentType	Determine if the message part matches the specified content type
MimeLocalizeText	Localize Unicode text to ANSI using a specific character set
MimeNormalizeText	Normalize Unicode text, eliminating confusable homoglyphs and symbols
MimeOpenMessage	Open and parse the specified text file as a MIME formatted message
MimeOpenMessageStore	Open the current message storage file
MimeParseAddress	Parse the specified email address
MimeParseDate	Parse the specified date and time, returning a 32-bit time value
MimeParseDate64	Parse the specified date and time, returning a 64-bit time value
MimeParseHeader	Parse the specified text and add to message header
MimeParseText	Parse the specified text and add to the current message
MimePurgeMessageStore	Purge all deleted messages from the current message store
MimeReplaceStoredMessage	Replace the current message in the message store
MimeResetMessage	Clear the current message, deleting all message parts
MimeSetExportOptions	Specify a bitmask that describes current message export options
MimeSetFileContentType	Set the content type for a specific file name extension
MimeSetLastError	Set the last error code

MimeSetMessageDate	Set the current date in the header for the current message
MimeSetMessageHeader	Create or update a header field in the current message
MimeSetMessageHeaderEx	Create or update a header field, with additional options
MimeSetMessagePart	Set the current message part index for the current message
MimeSetMessageText	Create or update the current message body
MimeSetMessageVersion	Set the MIME version for the current message
MimeStoreMessage	Store the current message in a message store
MimeUninitialize	Terminate use of the library by the application

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeAddMessageHeaders Function

```
BOOL WINAPI MimeAddMessageHeaders(
   HMESSAGE hMessage,
   LPCTSTR lpszHeaderList
);
```

The **MimeAddMessageHeaders** function adds one or more headers to the specified message.

Parameters

hMessage

Handle to the message.

lpszHeaderList

A null-terminated string which specifies one or more header values which should be set for the specified message. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the client handle is invalid, the function returns a value of zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeAddMessageHeaders** function enables your application to set one or more header values by providing a list of name/value pairs separated by a colon. Multiple header values may be provided by separating them with a newline character. This function is similar to calling the **MimeSetMessageHeader** function for each value. When the list of header values is parsed, extraneous whitespace is ignored; however, if the header list contains invalid text (for example, a missing colon separating a header name from its value) the function will fail and an error will be returned.

This function will only add or update header values in the main header block for the message. It cannot be used to update header values in a specific section of a multipart message. If you need to add or change a header value in a specific part of the message, use the **MimeSetMessageHeaderEx** function.

When adding custom, application-specific header values you should always prefix them with "X-" to avoid conflicting with standard headers.

Example

```
// Define a list of header values which should be included with the
// main header block for the message
LPCTSTR lpszHeaderList = _T("X-App-Sender: someuser@domain.tld\n") \
                         _T("X-App-Version: 1.5\n") \
                         _T("X-App-Mailer: AppMail (Win64)\n")

if (!MimeAddMessageHeaders(hMessage, lpszHeaderList))
{
    // Unable to add headers for this message
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageHeaders, MimeGetMessageHeader, MimeSetMessageHeader, MimeSetMessageHeaderEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeAppendMessageText Function

```
LONG WINAPI MimeAppendMessageText(
    HMESSAGE hMessage,
    LPCTSTR lpszText
);
```

The **MimeAppendMessageText** function appends the specified text to the body of the current message part.

Parameters

hMessage

Handle to the message.

lpszText

A null-terminated string which specifies the text to be appended to the current message part.

Return Value

If the function succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be appended to the current message part. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeClearMessageText, MimeCompareMessageText, MimeGetMessageText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeAttachData Function

```
BOOL WINAPI MimeAttachData(
   HMESSAGE hMessage,
   LPBYTE lpBuffer,
   LONG cbBuffer,
   LPCTSTR lpszContentName,
   LPCTSTR lpszContentType,
   DWORD dwOptions
);
```

The **MimeAttachData** function attaches the contents of the buffer to the message.

Parameters

hMessage

Handle to the message.

lpBuffer

Pointer to a byte buffer which contains the data to be attached to the message. This parameter may be NULL, in which case no data is attached, but an additional empty message part will be created.

cbBuffer

An unsigned integer which specifies the number of bytes of data in the buffer pointed to by the *lpBuffer* parameter. If the *lpBuffer* parameter is NULL, this value must be zero.

lpszContentName

Pointer to a string which specifies a name for the data being attached to the message. This typically is used as a file name by the mail client to store the data in. If this parameter is NULL or an empty string then no name is defined and the data is attached as inline content. Note that if a file name is specified with a path, only the base name will be used.

lpszContentType

Pointer to a string which specifies the type of data being attached. The value must be a valid MIME content type. If this parameter is NULL or an empty string, then the buffer will be examined to determine what kind of data it contains. If there is only text characters, then the content type will be specified as "text/plain". If the buffer contains binary data, then the content type will be specified as "application/octet-stream", which is appropriate for any type of data.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_ATTACH_DEFAULT	The data encoding is based on the content type. Text data is not encoded, and binary data is encoded using the standard base64 encoding algorithm.
MIME_ATTACH_BASE64	The data is always encoded using the standard base64 algorithm, even if the buffer only contains printable text characters.
MIME_ATTACH_UUCODE	The data is always encoded using the uuencode algorithm, even if the buffer only contains printable text characters.

MIME_ATTACH_QUOTED	The data is always encoded using the quoted-printable algorithm. This encoding should only be used if the data contains 8-bit text characters.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeAttachData** function allows an application to attach data to the message as either a file attachment or as inline content. The recipient of the message will see the attached data in the same way that they would see a file attached to the message using the **MimeAttachFile** function.

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeExportMessage, MimeExtractFile, MimeGetAttachedFileName, MimeGetFileContentType, MimeImportMessage, MimeSetFileContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeAttachFile Function

```
BOOL WINAPI MimeAttachFile(
   HMESSAGE hMessage,
   LPCTSTR lpszFileName,
   DWORD dwOptions
);
```

The **MimeAttachFile** function attaches the specified file to the message.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the file to be attached to the message.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_ATTACH_DEFAULT	The file attachment encoding is based on the file content type. Text files are not encoded, and binary files are encoded using the standard base64 encoding algorithm. This is the default option for file attachments.
MIME_ATTACH_BASE64	The file attachment is always encoded using the standard base64 algorithm, even if the attached file is a plain text file.
MIME_ATTACH_UUCODE	The file attachment is always encoded using the uuencode algorithm, even if the attached file is a plain text file.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach data that is stored in a memory buffer rather than a file, use the **MimeAttachData** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachData, MimeExportMessage, MimeExtractFile, MimeGetAttachedFileName, MimeGetFileContentType, MimeImportMessage, MimeSetFileContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeAttachImage Function

```
BOOL WINAPI MimeAttachImage(
   HMESSAGE hMessage,
   LPCTSTR lpszImageFile,
   LPCTSTR lpszContentId
);
```

The **MimeAttachImage** function attaches the specified file to the message as an inline image.

Parameters

hMessage

Handle to the message.

lpszImageFile

Pointer to a string which specifies the name of the image file to be attached to the message. This parameter cannot be NULL and must specify the name of an existing file.

lpszContentId

A null-terminated string which specifies the content ID associated with the inline image. If this parameter is NULL, a random content ID string will be automatically generated.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeAttachImage** function enables an application to attach an inline image to the message. Unlike a normal file attachment, this function is designed to be used with HTML formatted email messages that display images attached to the message. The ***lpszContentId*** parameter specifies the content ID string that is used with the HTML image tag to reference that image.

If the specified message is not a multipart message, it is marked as multipart and the attached image file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach regular files to the message, use the **MimeAttachFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachData, MimeAttachFile, MimeComposeMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCheckAuthenticationResults Function

```
BOOL WINAPI MimeCheckAuthenticationResults(
    HMESSAGE hMessage,
    DWORD dwAuthFlags
);
```

The **MimeCheckAuthenticationResults** function checks if the authentication status in the message matches one or more values.

Parameters

hMessage

A handle to the message.

dwAuthFlags

A bitmask that specifies one or more authentication flags to check for. These can be any combination of the values listed below.

Value	Description
MIME_AUTH_SPF_PASSED	The message passed the Sender Policy Framework (SPF) check. The sender's IP address was authorized to send mail for the domain in the envelope sender address.
MIME_AUTH_SPF_FAILED	The message failed the SPF check. The sending mail server is not authorized by the domain's SPF record to send mail on its behalf.
MIME_AUTH_DKIM_PASSED	The message passed the DomainKeys Identified Mail (DKIM) check. The signature applied by the sender was successfully verified using the domain's public key.
MIME_AUTH_DKIM_FAILED	The message failed the DKIM check. The signature could not be verified or did not match the message content, suggesting it may have been altered.
MIME_AUTH_DMARC_PASSED	The message passed DMARC evaluation. The message aligned with a valid SPF or DKIM result based on the sender domain's policy.
MIME_AUTH_DMARC_FAILED	The message failed DMARC evaluation. It did not pass SPF or DKIM checks in a way that aligns with the domain's published policy.
MIME_AUTH_TEMP_ERROR	A temporary error occurred while evaluating one or more authentication mechanisms. This may indicate a DNS lookup timeout or other transient failure.
MIME_AUTH_PERM_ERROR	A permanent error occurred during authentication checks. This may indicate a malformed DNS record or invalid message structure that prevents proper evaluation.

Return Value

This function returns a non-zero value if one or more of the specified flags match the values found in the message's `Authentication-Results` header. If there is no match, the function returns zero. This

function explicitly clears the last error code to assist in differentiating between an actual error condition (such as an invalid message handle) and a case where no authentication data matched.

Remarks

This function provides a convenient way to check if the authentication status of the message matches one or more values. In addition to the flags specified above, the following predefined flag combinations are also available:

- MIME_AUTH_PASSED is a combination of all passing results (SPF, DKIM, DMARC).
- MIME_AUTH_FAILED is a combination of all failing results.
- MIME_AUTH_ERROR is a combination of temporary or permanent errors.

This function was included as a convenience for languages other than C/C++, where unpacking bitfields returned by **MimeGetAuthenticationResults** may be more cumbersome. It allows for quick checks against common authentication outcomes using predefined flags. For example, to check whether the message failed authentication or encountered a temporary error, you could specify `MIME_AUTH_FAILED | MIME_AUTH_TEMP_ERROR`.

For more information about SPF, DKIM and DMARC, please refer to the **MimeGetAuthenticationResults** function.

Example

```
HMESSAGE hMessage = MimeOpenMessage(lpszFileName, MIME_OPEN_DEFAULT);

if (hMessage != INVALID_MESSAGE)
{
    // Check if any of the SPF, DKIM, or DMARC checks failed
    if (MimeCheckAuthenticationResults(hMessage, MIME_AUTH_FAILED))
        _tprintf(_T("This message has failed one or more authentication
checks\n"));

    MimeCloseMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeGetAuthenticationResults, MimeGetMessageHeader, MimeGetMessageHeaderEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeClearMessageText Function

```
BOOL WINAPI MimeClearMessageText(
    HMESSAGE hMessage
);
```

The **MimeClearMessageText** function deletes the text from the body of the current message part.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeAppendMessageText, MimeCompareMessageText, MimeGetMessageText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCloseMessage Function

```
BOOL WINAPI MimeCloseMessage(
   HMESSAGE hMessage
);
```

The **MimeCloseMessage** function closes the specified message and releases the memory allocated for the header and body.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCloseMessage** function closes the handle returned by the **MimeComposeMessage**, **MimeCreateMessage** or **MimeOpenMessage** functions. Once the message has been closed, the handle is invalid and should not be referenced. When your application no longer needs to access the contents of the message, this function must be called to release the HMESSAGE handle otherwise there will be a memory leak.

It is important to note that a message handle is not a kernel object and should never be closed with the **CloseHandle** function in the Windows API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeComposeMessage, MimeCreateMessage, MimeDeleteMessagePart, MimeOpenMessage, MimeResetMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCloseMessageStore Function

```
BOOL WINAPI MimeCloseMessageStore(
   HMESSAGESTORE hStorage,
   BOOL bPurgeMessages
);
```

The **MimeCloseMessageStore** function closes the specified message store.

Parameters

hStorage

Handle to the message store.

bPurgeMessages

An integer value which specifies if deleted messages are purged from the message store. A non-zero value specifies that all messages marked for deletion will be removed from the message store. A value of zero specifies that deleted messages will not be removed from the store.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCloseMessageStore** function closes the storage file that was previously opened, releasing all of the memory allocated for the message store and optionally purging all deleted messages. This function must be called when the application has finished accessing the messages in the message store. Failure to call this function will leave the storage file open and potentially locked by the process.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCopyMessageStore, MimeOpenMessageStore, MimePurgeMessageStore

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCompareMessageText Function

```
BOOL WINAPI MimeCompareMessageText(
    HMESSAGE hMessage,
    LONG *lpnOffset,
    LPCTSTR lpszText,
    LONG cchText,
    BOOL bCaseSensitive
);
```

The **MimeCompareMessageText** function compares a text string against the contents of the current message part.

Parameters

hMessage

Handle to the message.

lpnOffset

Pointer to a long integer which specifies the offset in the message at which to begin the comparison. This value will be updated when the function returns to indicate the offset position in the message where the comparison ended.

lpszText

Pointer to a string buffer which contains the text that is to be compared against the body of the message.

cchText

The number of characters in the buffer that should be compared against the body of the message.

bCaseSensitive

Boolean flag which specifies that the comparison should be case sensitive.

Return Value

If the text buffer matches the contents of the current message body, the function will return a non-zero value, and the *lpnOffset* argument will be set to position in the buffer where the match terminated. If the text buffer does not match, the function will return a value of zero, and the *lpnOffset* argument will be set to the position of the first non-matching character. The function will also return zero if one of the arguments is invalid. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAppendMessageText, MimeClearMessageText, MimeGetMessageText, MimeIsContentType MimeSetMessageText,

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeComposeMessage Function

```
HMESSAGE WINAPI MimeComposeMessage(
   LPCTSTR lpszFrom,
   LPCTSTR lpszTo,
   LPCTSTR lpszCc,
   LPCTSTR lpszSubject,
   LPCTSTR lpszMessageText,
   LPCTSTR lpszMessageHTML,
   UINT nCharacterSet,
   UINT nEncodingType
);
```

The **MimeComposeMessage** function creates a new message using the specified parameters.

Parameters

lpszFrom

A null-terminated string which specifies the sender's email address. This parameter may be NULL, in which case no sender address will be included in the message header.

lpszTo

A null-terminated string which specifies one or more recipient addresses. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no recipient addresses will be included in the message header.

lpszCc

A null-terminated string which specifies one or more addresses that will receive a copy of the message in addition to the listed recipients. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no carbon-copy addresses will be included in the message header.

lpszSubject

A null-terminated string which specifies the subject of the message. This parameter may be NULL, in which case no subject will be included in the message.

lpszMessageText

A null-terminated string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this parameter is NULL or points to an empty string, then the message will have an empty body unless the ***lpszMessageHTML*** parameter is not NULL.

lpszMessageHTML

A null-terminated string which contains the message using HTML formatting. If the ***lpszMessageText*** parameter is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the ***lpszMessageText*** argument is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

nCharacterSet

A numeric identifier which specifies the character set to use when composing the message. A value of zero specifies that the default UTF-8 character set should be used. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is INVALID_MESSAGE. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeComposeMessage** function composes a new message and returns a handle which can be used to further modify or export the message. To create an empty message without any predefined header values, call the **MimeCreateMessage** function.

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* parameter cannot specify multiple addresses, however it is permitted with the *lpszTo* and *lpszCc* parameters.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Call the **MimeCloseMessage** function to free the memory allocated for this message when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCloseMessage, MimeCreateMessage, MimeCreateMessagePart, MimeExportMessage, MimeImportMessage, MimeInitialize, MimeOpenMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCopyMessageStore Function

```
BOOL WINAPI MimeCopyMessageStore(
   HMESSAGESTORE hStorage,
   LPCTSTR lpszFileName
);
```

The **MimeCopyMessageStore** function duplicates the contents of the specified message store in a new file.

Parameters

hStorage

Handle to the message store.

lpszFileName

A null-terminated string which specifies the name of the file that the messages will be copied to. This parameter cannot be NULL and must specify a valid file path and name.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCopyMessageStore** function is used to create a copy of the specified message store in a new file. If the file does not exist, it will be created. If the file already exists, then the contents will be overwritten with the contents of the message store.

Messages that have been marked for deletion are not copied to the new message store file.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCloseMessageStore, MimeOpenMessageStore, MimePurgeMessageStore

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCreateMessage Function

```
HMESSAGE MimeCreateMessage();
```

The **MimeCreateMessage** function creates a new empty message.

Parameters

There are no parameters.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is INVALID_MESSAGE. To get extended error information, call **MimeGetLastError**.

Remarks

To create a message with defined values for the sender, recipient, subject and body it is recommended that you use the **MimeComposeMessage** function.

Call the **MimeCloseMessage** function to free the memory allocated for this message when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCloseMessage, MimeComposeMessage, MimeCreateMessageEx, MimeCreateMessagePart, MimeExportMessage, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCreateMessageEx Function

```
HMESSAGE WINAPI MimeCreateMessageEx(
    DWORD dwImportMode,
    DWORD dwImportOptions,
    LPVOID lpvMessage,
    DWORD dwMessageSize
);
```

The **MimeCreateMessageEx** function creates a new message and imports the contents from a file or memory.

Parameters

dwImportMode

An unsigned integer which specifies how the message contents will be imported. It may be one of the following values:

Value	Description
MIME_IMPORT_DEFAULT	The default import mode. If the ***lpvMessage*** parameter is NULL, the function will return a handle to an empty message. Otherwise, the ***lpvMessage*** parameter is a pointer to a string which specifies the name of a file that contains the message. The ***dwMessageSize*** parameter is ignored.
MIME_IMPORT_FILE	The ***lpvMessage*** parameter is a pointer to a string which specifies the name of a file that contains the message. If the file does not exist, or is not a regular text file, an error will occur. The ***dwMessageSize*** parameter is ignored.
MIME_IMPORT_CLIPBOARD	The contents of the message is imported from the system clipboard. The ***lpvMessage*** parameter is ignored. The ***dwMessageSize*** parameter is ignored.
MIME_IMPORT_MEMORY	The contents of the message is imported from a local buffer. The ***lpvMessage*** parameter must point to a byte array which contains the message to be imported. The ***dwMessageSize*** parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.
MIME_IMPORT_HGLOBAL	The contents of the message is imported from a global memory buffer. The ***lpvMessage*** parameter must be a global memory handle which contains the message. The ***dwMessageSize*** parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.

dwImportOptions

An unsigned integer which specifies how the message will be imported:

Value	Description

MIME_OPTION_DEFAULT	The default import options. Currently this is the only valid value for this parameter and applications should always specify this constant.

lpvMessage

A null-terminated string, a byte buffer or a global memory handle. The ***dwImportMode*** parameter determines how this pointer is used by the function.

dwMessageSize

An unsigned integer value which specifies the size of the message to import. This parameter is only used when importing a message from a memory buffer. The message size is determined automatically when the message is imported from a file or the system clipboard.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is INVALID_MESSAGE. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCreateMessageEx** function creates a new message from the contents of a file or memory, and returns a handle to the parsed message. When the message is no longer needed by the application, it should call **MimeCloseMessage** to free the memory allocated for the message and release the handle.

To create a new message with predefined header values and content, use the **MimeComposeMessage** function.

Example

```
// Creates a message using an HGLOBAL handle which references
// a block of memory containing an email message
hMessage = MimeCreateMessageEx(MIME_IMPORT_HGLOBAL,
                               MIME_OPTION_DEFAULT,
                               (LPVOID)hgblMessage,
                               0);

if (hMessage != INVALID_MESSAGE)
{
    // The message has been successfully created
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCloseMessage, MimeComposeMessage, MimeCreateMessage, MimeExportMessage, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCreateMessagePart Function

```
INT WINAPI MimeCreateMessagePart(
    HMESSAGE hMessage
);
```

The **MimeCreateMessagePart** function creates a new part for the specified message. If this the first part created for a message that does not have the multipart content type specified, the message is marked as multipart and the header fields are updated.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the new message part number. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeAttachFile, MimeCreateMessage, MimeDeleteMessagePart, MimeGetMessagePart, MimeGetMessagePartCount, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeCreateMessagePartEx Function

```
INT WINAPI MimeCreateMessagePartEx(
    HMESSAGE hMessage,
    UINT nCharacterSet,
    UINT nEncodingType,
    LPCTSTR lpszText
);
```

The **MimeCreateMessagePartEx** function creates a new part for the specified message. If this the first part created for a message that does not have the multipart content type specified, the message is marked as multipart and the header fields are updated.

Parameters

hMessage

Handle to the message.

nCharacterSet

A numeric identifier which specifies the character set to use when composing the message. A value of zero specifies the character set should be the same character set used to initially compose the message. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use Latin character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or

	when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

lpszText

A null-terminated string which specifies the text to be included in the body of the new message part. If this parameter is NULL or points to an empty string, no text is added to the message part.

Return Value

If the function succeeds, the return value is the new message part number. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeComposeMessage, MimeCreateMessage, MimeCreateMessagePart, MimeDeleteMessagePart, MimeGetMessagePart, MimeGetMessagePartCount, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeDecodeText Function

```
LONG WINAPI MimeDecodeText(
   UINT nEncodingType,
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG cchOutput
);
```

The **MimeDecodeText** function decodes a string which was previously encoded using base64 or quoted-printable encoding.

Parameters

nEncodingType

An integer value that specifies the encoding method used. It may be zero or one of the following values:

Value	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this function.

lpszInput

A null-terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be decoded. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpszOutput

A null-terminated string buffer that will contain the decoded text. This buffer must be large enough to store all of the characters in the decoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the decoded text and

terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully decoded, the return value is the length of the decoded output string. If the function returns zero, then no text was decoded and the output string buffer will be empty. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this function because the message parser will automatically decode the message text if necessary.

This function and the **MimeEncodeText** function use the UTF-8 character set. If the Unicode version of this function is called, the output text will be decoded as UTF-8, then converted to UTF-16 and returned to the caller. If the ANSI version of this function is used, the decoded output will always be returned to the caller using the UTF-8 character set.

If an unsupported encoding type is specified, this function will return MIME_ERROR and the output text string will be empty. In most cases, it is preferable to use MIME_ENCODING_BASE64 as the encoding method, with quoted-printable encoding only used for legacy support.

If the original text was encoded using a different character set, use the **MimeDecodeTextEx** function, which enables you to specify an alternate character set.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeDecodeTextEx, MimeEncodeText, MimeEncodeTextEx, MimeGetMessageText, MimeLocalizeText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeDecodeTextEx Function

```
LONG WINAPI MimeDecodeTextEx(
   UINT nCharacterSet,
   UINT nEncodingType,
   DWORD dwReserved,
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG cchOutput
);
```

The **MimeDecodeTextEx** function decodes a string which was previously encoded using base64 or quoted-printable encoding. This extended version of the function enables the caller to specify the character set for the original text.

Parameters

nCharacterSet

A numeric identifier which specifies the character set to use when decoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is decoded. If this value does not match the character set used when the text was originally encoded, the resulting output text may be invalid.

nEncodingType

An integer value that specifies the encoding method used. It may be zero or one of the following values:

Value	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this function.

dwReserved

An integer value reserved for internal use. This value must always be zero.

lpszInput

A null-terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be decoded. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpszOutput

A null-terminated string buffer that will contain the decoded text. This buffer must be large enough to store all of the characters in the decoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the decoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully decoded, the return value is the length of the decoded output string. If the function returns zero, then no text was decoded and the output string buffer will be empty. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this function because the message parser will detect which character set and encoding was used, then automatically decode the message text if necessary.

The value of the ***nCharacterSet*** parameter does not affect the resulting output text, it is only used when decoding the input text. If the Unicode version of this function is called, the output text will be converted to UTF-16 and returned to the caller. If the ANSI version of this function is used, the decoded output will always be returned to the caller using the UTF-8 character set.

If the ***nCharacterSet*** parameter is specified as MIME_CHARSET_UTF16, the encoding type must be MIME_ENCODING_BASE64. Other encoding methods are not supported for Unicode strings and will cause the function to fail. In most cases, it is preferable to use MIME_ENCODING_BASE64 as the encoding method, with quoted-printable encoding only used for legacy support.

If an unsupported encoding type is specified, this function will return MIME_ERROR and the output text string will be empty. This function cannot be used to decode uuencoded text.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeDecodeText, MimeEncodeText, MimeEncodeTextEx, MimeGetMessageText, MimeLocalizeText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeDeleteMessageHeader Function

```
BOOL WINAPI MimeDeleteMessageHeader(
    HMESSAGE hMessage,
    LPCTSTR lpszHeader
);
```

The **MimeDeleteMessageHeader** function deletes the specified header field from the message.

Parameters

hMessage

Handle to the message.

lpszHeader

Pointer to a string which specifies the header field that will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetMessageHeader, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeDeleteMessagePart Function

```
BOOL WINAPI MimeDeleteMessagePart(
    HMESSAGE hMessage,
    INT nMessagePart
);
```

The **MimeDeleteMessagePart** function deletes the specified message part from the multipart message. The memory allocated for the message part is released.

Parameters

hMessage

Handle to the message.

nMessagePart

The message part index to delete.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function cannot be used to delete part zero, which is the main body of the message. Instead use the **MimeResetMessage** function to clear the entire message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCreateMessagePart, MimeCreateMessagePartEx, MimeGetMessagePart, MimeGetMessagePartCount, MimeResetMessage, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeDeleteStoredMessage Function

```
BOOL WINAPI MimeDeleteStoredMessage(
   HMESSAGESTORE hStorage,
   LONG nMessageId
);
```

The **MimeDeleteStoredMessage** function removes the specified message from the message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which identifies the message that is to be removed from the message store. Message numbers begin at one and increment for each message in the store.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeDeleteStoredMessage** function marks the specified message for deletion from the storage file. When the message store is closed or purged, the message is removed from the file. Once a message has been marked for deletion, it may no longer be referenced by the application. For example, you cannot access the contents of a message that has been deleted.

The message store must be opened with write access. This function will fail if you attempt to delete a message from a storage file that has been opened for read-only access. If the application needs to delete messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCloseMessageStore, MimeFindStoredMessage, MimeGetStoredMessage, MimeGetStoredMessageCount, MimePurgeMessageStore

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeEncodeText Function

```
LONG WINAPI MimeEncodeText(
   UINT nEncodingType,
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG cchOutput
);
```

The **MimeEncodeText** function encodes a string using base64 or quoted-printable encoding.

Parameters

nEncodingType

An integer value that specifies the encoding method used. It may be zero or one of the following values:

Value	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this function.

lpszInput

A null-terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be encoded. If this value is -1, the entire length of the string up to the terminating null will be encoded.

lpszOutput

A null-terminated string buffer that will contain the encoded text. This buffer must be large enough to store all of the characters in the encoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the encoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully encoded, the return value is the length of the encoded output string. If the function returns zero, then no text was encoded and the output string buffer will be empty. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this function to encode text when using the **MimeSetMessageText** function. The library will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

This function and the **MimeDecodeText** function use the UTF-8 character set. If the Unicode version of this function is called, the input text will be converted to UTF-8 and then encoded. If the ANSI version of this function is used, the input text is expected to be UTF-8 or ASCII text. If you are encoding text which uses non-ASCII characters, it is recommended that you use the Unicode version of this function.

If an unsupported encoding type is specified, this function will return MIME_ERROR and the output text string will be empty. This function cannot be used to create uuencoded text. In most cases, it is preferable to use MIME_ENCODING_BASE64 as the encoding method, with quoted-printable encoding only used for legacy support.

If you want the text to be encoded using a different character set, you can use the **MimeEncodeTextEx** function. In most cases, your application should use the default UTF-8 character set when encoding and decoding text which contains non-ASCII characters, rather than using specific character sets.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeDecodeText, MimeDecodeTextEx, MimeEncodeTextEx, MimeGetMessageText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeEncodeTextEx Function

```
LONG WINAPI MimeEncodeTextEx(
   UINT nCharacterSet,
   UINT nEncodingType,
   DWORD dwReserved,
   LPCTSTR lpszInput,
   LONG cchInput,
   LPTSTR lpszOutput,
   LONG cchOutput
);
```

The **MimeEncodeTextEx** function encodes a string using base64 or quoted-printable encoding. This extended version of the function enables the caller to specify the character set for the input text.

Parameters

nCharacterSet

A numeric identifier which specifies the character set to use when encoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is encoded.

nEncodingType

An integer value that specifies the encoding method used. It may be one of the following values:

Value	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

dwReserved

An integer value reserved for internal use. This value must always be zero.

lpszInput

A null-terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be encoded. If this value is -1, the entire length of the string up to the terminating null will be encoded.

lpszOutput

A null-terminated string buffer that will contain the encoded text. This buffer must be large enough to store all of the characters in the encoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the encoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully encoded, the return value is the length of the encoded output string. If the function returns zero, then no text was encoded and the output string buffer will be empty. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this function to encode text when using the **MimeSetMessageText** function. The library will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

If the ***nCharacterSet*** parameter parameter is non-zero, the function will encode the text using the specified character set. If the Unicode version of this function is called, the input text is converted to ANSI using the code page associated with the character set. If the ANSI version of this function is called, the input text is converted to Unicode using the system default code page, and then back to ANSI using the specified character set.

If the ***nCharacterSet*** parameter specifies the MIME_CHARSET_UTF16 character set, you must specify MIME_ENCODING_BASE64 as the encoding method. Other encoding methods are not supported for Unicode strings and will cause the function to fail. It is not recommended you encode text as UTF-16 unless there is a specific requirement to use that character set.

It is recommended that you use the MIME_CHARSET_UTF8 character set whenever possible. It is capable of encoding all Unicode code points, and is a standard for virtually all modern Internet applications. In most cases, it is preferable to use MIME_ENCODING_BASE64 as the encoding method, with quoted-printable encoding only used for legacy support.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeDecodeText, MimeEncodeText, MimeEncodeTextEx, MimeGetMessageText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeEnumAttachments Function

```
INT WINAPI MimeEnumAttachments(
   HMESSAGE hMessage,
   LPCTSTR* lpFileNames,
   INT nMaxFiles
);
```

The **MimeEnumAttachments** function enumerates all of the file attachments in the specified message.

Parameters

hMessage

Handle to the message.

lpFileNames

A pointer to a an array of null-terminated strings that contain the names of the files attached to the message. If this parameter is NULL, the function will only return the number of files attached to the message.

nMaxFiles

An integer value that specifies the maximum size of the array of string pointers specified by the *lpFileNames* parameter. If this value is zero, the *lpFileNames* parameter is ignored and the function will only return the number of files attached to the message.

Return Value

If the function succeeds, the return value is the number of files attached to the message. If the message does not contain any file attachments, this function will return a value of zero. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function can be used to obtain the names of the files attached to the message. If this function is used with languages other than C/C++, it is important to note that the *lpFileNames* parameter is not a pointer to a string, but rather a pointer to an array of pointers to null-terminated strings. It is not recommended that you use this function with languages that do not support C-style arrays. For example, Visual Basic represents arrays as SAFEARRAY structures and is not compatible with this function.

Example

```
LPCTSTR lpszFiles[MAXFILES];

INT nFiles = MimeEnumAttachments(hMessage, lpszFiles, MAXFILES);
if (nFiles == MIME_ERROR)
{
    DWORD dwError = MimeGetLastError();
    _tprintf(_T("Unable to enumerate attachments, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("There are %d files attached to the message\n"), nFiles);
for (INT nIndex = 0; nIndex < nFiles; nIndex++)
{
    BOOL bResult = MimeExtractFileEx(hMessage, -1, lpszFiles[nIndex], NULL, 0);
```

```
    if (bResult)
        _tprintf(_T("%d: extracted attachment \"%s\"\n"), nIndex,
lpszFiles[nIndex]);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeExportMessage, MimeExtractAllFiles, MimeExtractFileEx, MimeGetAttachedFileName, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeEnumMessageHeaders Function

```
INT WINAPI MimeEnumMessageHeaders(
   HMESSAGE hMessage,
   LPCTSTR *lpHeaderList,
   INT nMaxHeaders
);
```

The **MimeEnumMessageHeaders** function returns a list of pointers to all header field names in the current message part. This can be used in conjunction with the **MimeGetMessageHeader** function to retrieve the values for every header in the message.

Parameters

hMessage

Handle to the message.

lpHeaderList

Pointer to an array of pointers to null-terminated header field names. If this parameter is NULL, the function only returns the number of headers in the current message part.

nMaxHeaders

The maximum number of header fields which may be returned in the ***lpHeaderList*** parameter.

Return Value

If the function succeeds, the return value is the total number of headers that are defined in the current message part. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The values returned in the header list array must not be directly modified by the application. There is no specific order in which the header fields are enumerated by this function. The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

Example

```
// Determine the total number of headers in the current
// message part

nHeaders = MimeEnumMessageHeaders(hMessage, NULL, 0);
if (nHeaders > 0)
{

    // Allocate memory for the list of headers

    lpHeaderList = (LPCTSTR *)LocalAlloc(LPTR, nHeaders * sizeof(LPCTSTR));
    assert(lpHeaderList != NULL);

    // Retrieve the list of headers in the current
    // message part, and get their values

    MimeEnumMessageHeaders(hMessage, lpHeaderList, nHeaders);
    for (nIndex = 0; nIndex < nHeaders; nIndex++)
```

```
    {

        LPCTSTR lpszValue;
        lpszValue = MimeGetMessageHeader(hMessage, lpHeaderList[nIndex]);
        assert(lpszValue != NULL);

        printf("%s: %s\n", lpHeaderList[nIndex], lpszValue);

    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetNextMessageHeader, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeEnumMessageRecipients Function

```
INT WINAPI MimeEnumMessageRecipients(
   HMESSAGE hMessage,
   LPCTSTR lpszExtraAddress,
   LPTSTR lpBuffer,
   LPDWORD lpcchBuffer
);
```

The **MimeEnumMessageRecipients** function returns a null-terminated list of strings which contain the email address of each recipient for the specified message.

Parameters

hMessage

Handle to the message.

lpszExtraAddress

A null-terminated string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpBuffer

Pointer to buffer which will contain zero or more null-terminated strings. The end of the string list is indicated by an additional terminating null. If this parameter is NULL, the function will calculate the minimum number of bytes required to store the addresses and return the value in the ***lpcbBuffer*** parameter.

lpcchBuffer

A pointer to an unsigned integer which should be initialized to the maximum number of characters that can be copied into the buffer specified by the ***lpBuffer*** parameter. When the function returns, it will be updated to contain the actual number of characters copied into the buffer. If the ***lpBuffer*** parameter is NULL, then this value will contain the minimum number of characters required to store all of the recipient addresses in the current message.

Return Value

If the function succeeds, the return value is the total number of recipients for the current message. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeEnumMessageRecipients** function returns a list of recipient email addresses for the specified message, with each address being terminated by a null character. The end of the list is indicated by an additional null character. To determine the size of the buffer you should pass to this function, you can specify the ***lpBuffer*** parameter as NULL and initialize the value of the ***lpcchBuffer*** parameter to zero.

This function is primarily designed for use with C/C++ and languages that can easily use C-style strings and pointers to string values. In many cases, it may be preferable to use the **MimeGetAllRecipients** function which returns a comma-separated list of recipient addresses in a string buffer.

If you are using the Unicode version of this function, keep in mind that the ***lpcchBuffer*** parameter

returns the number of characters in the output buffer, not the number of bytes. If you are dynamically allocating the buffer, make sure you allocate enough memory to store Unicode strings, which are two bytes per character.

Example

```
LPTSTR lpRecipients = NULL;
DWORD cchRecipients = 0;
INT nRecipients = 0;

// Determine the number of characters that should be allocated to store
// all of the recipient addresses in the current message

nRecipients = MimeEnumMessageRecipients(hMessage,
                                        NULL,
                                        NULL,
                                        &cchRecipients);

// Allocate the memory for the string buffer that will contain all
// of the recipient addresses and call MimeEnumMessageRecipients
// again to store those addresses in the buffer

if (nRecipients > 0 && cchRecipients > 0)
{
    lpRecipients = (LPTSTR)LocalAlloc(LPTR, cchRecipients * sizeof(TCHAR));
    if (lpRecipients == NULL)
        return; // Virtual memory exhausted

    nRecipients = MimeEnumMessageRecipients(hMessage,
                                            NULL,
                                            lpRecipients,
                                            &cchRecipients);
}

// Move through the buffer, processing each recipient address
// that was returned

if (nRecipients > 0)
{
    LPTSTR lpszAddress = lpRecipients;
    INT cchAddress;

    while (lpszAddress != NULL)
    {
        if ((cchAddress = lstrlen(lpszAddress)) == 0)
            break;

        // lpszAddress specifies a recipient address
        // Advance to the next address string in the buffer
        lpszAddress += cchAddress + 1;
    }
}

if (lpRecipients)
{
    LocalFree((HLOCAL)lpRecipients);
    lpRecipients = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetAllRecipients, MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetNextMessageHeader, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeExportMessage Function

```
BOOL WINAPI MimeExportMessage(
   HMESSAGE hMessage,
   LPCTSTR lpszFileName
);
```

The **MimeExportMessage** function writes the specified message to a file. If the file does not exist, it will be created. If it does exist, it will be overwritten with the contents of the message.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the file to be created from the message. If this parameter is NULL, the contents of the message will be copied to the system clipboard.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function is provided for backwards compatibility with previous versions of the API. To export the contents of the message to a buffer in memory, use the **MimeExportMessageEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCreateMessage, MimeExportMessageEx, MimeImportMessage, MimeImportMessageEx, MimeSetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeExportMessageEx Function

```
BOOL WINAPI MimeExportMessageEx(
   HMESSAGE hMessage,
   DWORD dwExportMode,
   DWORD dwExportOptions,
   LPVOID lpvMessage,
   LPDWORD lpdwMessageSize
);
```

The **MimeExportMessageEx** function exports the message to a file, the system clipboard or global memory buffer.

Parameters

hMessage

Handle to the message.

dwExportMode

An unsigned integer which specifies how the message contents will be exported. It may be one of the following values:

Value	Description
MIME_EXPORT_DEFAULT	The default export mode. If the ***lpvMessage*** parameter is NULL, then the contents of the message will be copied to the system clipboard. Otherwise, the ***lpvMessage*** parameter is a pointer to a string which specifies the name of a file to store the message in.
MIME_EXPORT_FILE	The ***lpvMessage*** parameter is a pointer to a null-terminated string which specifies the name of a file to store the message in. If the file does not exist, it will be created. Otherwise the file will be overwritten with the contents of the message.
MIME_EXPORT_CLIPBOARD	The contents of the message is copied to the system clipboard. The ***lpvMessage*** parameter is ignored.
MIME_EXPORT_MEMORY	The contents of the message is copied to a local buffer. The ***lpvMessage*** parameter must point to a byte array which will contain the message contents when the function returns. The ***lpdwMessageSize*** parameter must be initialized to the maximum size of the buffer and will contain the number of bytes copied to the buffer when the function returns.
MIME_EXPORT_HGLOBAL	The contents of the message is stored in a global memory buffer. The ***lpvMessage*** parameter must point to a global memory handle which will reference the message buffer when the function returns. The ***lpdwMessageSize*** parameter will contain the number of bytes allocated for the message. The client application is responsible for releasing the memory handle when the message contents are no longer needed.

dwExportOptions

An unsigned integer which specifies how the message will be exported. The following values may be combined using a bitwise Or operator:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

lpvMessage

A null-terminated string, a byte buffer or a global memory handle. The ***dwExportMode*** parameter determines how this pointer is used by the function.

lpdwMessageSize

A pointer to an unsigned integer value which will contain the size of the message when the function exits. This parameter may be NULL if you do not require this information, except if the ***dwExportMode*** parameter is MIME_EXPORT_MEMORY. In this case, the parameter must point to a value initialized with the maximum size of the byte buffer that has been passed to the function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

If the MIME_EXPORT_MEMORY mode is used to export the message to a pre-allocated buffer, the ***lpdwMessageSize*** parameter must be initialized to the size of the buffer before being passed to the function by reference. If this value is not properly initialized by the application, it can result in an error or a general protection exception. If the function succeeds, the message will be copied to the buffer. If the function fails, the previous contents of the buffer will not be preserved. It is not guaranteed that the buffer will be terminated with a null byte. If the buffer is not large enough to store the entire message, the function will fail. The **MimeGetMessageSize** function can be used to determine the

minimum size of the buffer required to store the message.

If the MIME_EXPORT_HGLOBAL mode is used, a global memory handle will be returned to the caller which contains the message contents. This handle must be dereferenced using the **GlobalLock** function. No changes should be made to this copy of the message. If you wish to modify the contents of the message, allocate a local buffer and copy the message contents, or use the MIME_EXPORT_MEMORY option instead. Your application is responsible for calling **GlobalUnlock** and **GlobalFree** to unlock and free the handle when it is no longer needed.

Example

```
// Exports the message contents to a global buffer
HGLOBAL hgblMessage = NULL;
DWORD dwMessageSize = 0;

bResult = MimeExportMessageEx(hMessage,
                              MIME_EXPORT_HGLOBAL,
                              MIME_OPTION_DEFAULT,
                              &hgblMessage,
                              &dwMessageSize);

if (bResult)
{
    LPBYTE lpMessage = (LPBYTE)GlobalLock(hgblMessage);

    if (lpMessage != NULL)
    {
        // Process the contents of the message
    }

    GlobalUnlock(hgblMessage);
    GlobalFree(hgblMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCreateMessage, MimeExportMessage, MimeGetMessageSize, MimeImportMessage, MimeImportMessageEx, MimeSetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeExtractAllFiles Function

```
INT WINAPI MimeExtractAllFiles(
   HMESSAGE hMessage,
   LPCTSTR lpszDirectory,
   DWORD dwReserved
);
```

The **MimeExtractAllFiles** function extracts all of the file attachments in a message and stores them in the specified directory.

Parameters

hMessage

Handle to the message.

lpszDirectory

A null-terminated string which specifies the name of the directory where the file attachments should be stored. If this parameter is NULL or points to an empty string, the attached files will be stored in the current working directory on the local system.

dwReserved

An unsigned integer value that is reserved for future use. This parameter must always have a value of zero.

Return Value

If the function succeeds, the return value is the number of file attachments which were extracted from the message. If the message does not contain any file attachments, this function will return a value of zero. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function will extract all of the files that are attached to the message and store them in the specified directory. The directory must exist and the current user must have the appropriate permissions to create files there. If a file with the same name as the attachment already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

To determine the file names for each of the attachments in a message, use the **MimeEnumAttachments** function. To store a file attachment on the local system using a name that is different than the file name of the attachment, use the **MimeExtractFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeEnumAttachments, MimeExportMessage, MimeExtractFileEx, MimeGetAttachedFileName, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeExtractFile Function

```
BOOL WINAPI MimeExtractFile(
   HMESSAGE hMessage,
   LPCTSTR lpszFileName
);
```

The **MimeExtractFile** function extracts a file attachment from the current message part and stores it on the local system.

Parameters

hMessage

Handle to the message.

lpszFileName

A null-terminated string which specifies the name of a file on the local system.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function will store the contents of a file attachment in the current message part to the specified file on the local system. If a path is specified as part of the file name, it must exist and the current user must have the appropriate permissions to create the file. If a file with the same name already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

To determine if the current message part contains a file attachment, use the **MimeGetAttachedFileName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeExportMessage, MimeExtractFileEx, MimeGetAttachedFileName, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeExtractFileEx Function

```
BOOL WINAPI MimeExtractFileEx(
   HMESSAGE hMessage,
   INT nMessagePart,
   LPCTSTR lpszAttachment,
   LPCTSTR lpszFileName,
   DWORD dwReserved
);
```

The **MimeExtractFileEx** function extracts a file attachment from the message and stores it on the local system.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value that specifies the message part that contains the file attachment. This value may be -1, in which case the current message part will be used, or the function will search for an attachment with a file name that matches the value of the ***lpszAttachment*** parameter.

lpszAttachment

A null-terminated string that specifies the file name for the attachment in the message. If the file name of the attachment is not known, this parameter can be NULL or point to an empty string. This parameter is ignored if the ***nMessagePart*** parameter has a value greater than -1.

lpszFileName

A null-terminated string that specifies the name of a file on the local system. If this parameter is NULL or points to an empty string, the value of the ***lpszAttachment*** parameter will specify the name of the file in the current working directory. If both the ***lpszAttachment*** and ***lpszFileName*** parameters are NULL, the function will fail.

dwReserved

An unsigned integer value that is reserved for future use. This parameter must always have a value of zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function will store the contents of a file attachment in the current message part to the specified file on the local system. If a path is specified as part of the file name, it must exist and the current user must have the appropriate permissions to create the file. If a file with the same name already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

If the ***nMessagePart*** parameter has a value of -1 and the ***lpszAttachment*** parameter specifies a file name, this function will search the entire message for an attachment with the same file name. The search is not case-sensitive, however it must match the attachment file name completely. This function will not match partial file names or names that include wildcard characters. If a match is found, the contents of that attachment will be stored in the file specified by the ***lpszFileName*** parameter.

If the *nMessagePart* parameter has a value of -1 and the *lpszAttachment* parameter is NULL or points to an empty string, then the attachment in the current message part will be stored in the specified file. If the current message part does not contain a file attachment, this function will fail. Calling this function in this manner is effectively the same as calling the **MimeExtractFile** function.

To extract all of the files attached to a message in a single function call, use the **MimeExtractAllFiles** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeEnumAttachments, MimeExportMessage, MimeExtractAllFiles, MimeExtractFile, MimeFindAttachment, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeFilterMessageText Function

```
LONG WINAPI MimeFilterMessageText(
   HMESSAGE hMessage,
   LPTSTR lpszFilteredText,
   LONG nMaxLength,
   DWORD dwOptions
);
```

The **MimeFilterMessageText** function filters the contents of a HTML or plain text message.

Parameters

hMessage

Handle to the message.

lpszFilteredText

A null-terminated string buffer which will contain the filtered text when the function returns. This buffer must be large enough to contain all of the filtered content and a terminating null character. If this parameter is NULL, the filtered text will not be copied. This can be used to determine the amount of filtered text that would be returned. This can be used to determine how much memory should be allocated for the filtered text; however, remember to allocate additional space for the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the maximum length is not large enough to store all of the filtered text and a terminating null character, the function will return an error. This value may be zero which indicates you want determine the length of the filtered text without copying any characters.

dwOptions

An unsigned integer which specifies one or more bit flags. which controls what content is filtered from the message. It may be one or more of the following values.

Value	Description
MIME_FILTER_DEFAULT	Perform default filtering which removes HTML markup and decodes HTML entities. No additional filtering of URLs or text is performed.
MIME_FILTER_REMOVE_QUERY	Remove query parameters from hyperlinks such as `?id=123`. Useful for stripping tracking information or session data.
MIME_FILTER_REMOVE_CREDENTIALS	Remove embedded user credentials in URLs such as `user:pass@example.com`, preventing obfuscation or accidental disclosure.
MIME_FILTER_REMOVE_LINKS	Remove all hyperlinks from the message content. This includes both anchor tags in HTML and plain text URLs.
MIME_FILTER_REMOVE_UNICODE	Strip all non-ASCII (Unicode) characters from the message. Use this when only 7-bit clean output is required.

MIME_FILTER_REJECT_LOCAL	Reject any link that refers to internal or local domains, including **.local**, **.lan**, and **localhost**.
MIME_FILTER_REJECT_IP	Reject links that use raw IP addresses instead of hostnames, such as **http://192.168.1.1**.
MIME_FILTER_REJECT_PUNYCODE	Reject hyperlinks using Punycode-encoded domain names, such as those with the **xn--** prefix. This helps prevent spoofing using visually similar characters.
MIME_FILTER_REJECT_HTTP	Reject any hyperlinks that use **http://** instead of the secure **https://** protocol.
MIME_FILTER_REJECT_UNSAFE	Reject links using unsafe schemes such as **file:**, **data:**, or **javascript:**. These links are excluded by default unless overridden.
MIME_FILTER_ALLOW_PUNYCODE	Explicitly permit Punycode domain names in URLs even when **MIME_FILTER_REMOVE_UNICODE** is specified. Use when legitimate IDN domains are expected.
MIME_FILTER_ALLOW_UNSAFE	Allow potentially unsafe schemes such as **file:** or **javascript:** in message content. Use with caution and only when the message source is fully trusted.
MIME_FILTER_ALLOW_SYMBOLS	Allow Unicode symbols such as emojis and pictographs in the filtered text.
MIME_FILTER_ALLOW_CONFUSABLES	Allow Unicode characters which can be confused with normal ASCII characters. Use only when the message source is fully trusted.

Return Value

The function will return the number of characters of filtered text copied to the string buffer, not including the terminating null character. If the ***lpszFilteredText*** parameter is NULL, the function will return the length of the filtered text without copying any characters. A return value of zero indicates the message part does not contain any message text. If the function fails it will return MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeFilterMessageText** function filters MIME-encoded message parts by removing HTML tags and other potentially unsafe or undesired content. It handles both HTML and plain text content types. For HTML, the function converts the markup to plain text and decodes HTML entities. For plain text, it scans for embedded URLs and applies filtering based on the specified options.

When filtering HTML markup and text from message content, this function can:

- Remove script, style, and other non-visible or unsafe elements.
- Convert common tag structures, such as lists and headers, into readable plain text.
- Extract and optionally filter or remove hyperlinks and image alt text.
- Decode common HTML entities such as **&trade** and **&copy**, as well as numeric entites like

`&#x20.`

- Normalize Unicode characters to their canonical values (NKFD normalization).
- Replace confusable homoglyhs with their ASCII counterparts.
- Remove symbols such as emojis and pictographs which can confuse pattern matching.

The function is designed to produce clean, human-readable output suitable for previewing message contents in text-only environments, logging, or performing further analysis. It ensures that embedded credentials, tracking links, or insecure resources are removed or sanitized based on caller preferences.

By default, the function does not remove Unicode characters or strip links unless explicitly instructed via flags. The `MIME_FILTER_STRICT` constant is a recommended combination for maximum safety and removes potentially malicious or misleading elements from messages. Characters that are visually indistinguishable from standard Latin letters, such as the Cyrillic "a" (U+0430) and Latin "a" (U+0061), will be converted to their ASCII equivalent, preventing homoglyhph attacks and attempts to obsfucate links or defeat pattern matching.

To obtain the original message content, use the **MimeGetMessageText** function. If you want to perform generalized normalization on the contents of a message, you can also use the **MimeNormalizeText** function. However, this function already performs equivalent normalization on the message text, so it is not necessary to use either **MimeNormalizeText** or the **NormalizeString** function on the output from this function.

Example

```
// Search a multipart message for the HTML version of the message
// and write the filtered content to standard output

HMESSAGE hMessage = MimeOpenMessage(pszFileName, MIME_OPEN_DEFAULT);

if (hMessage == INVALID_MESSAGE)
{
    _tprintf(_T("Unable to open message %s\n"), pszFileName);
    return;
}

// Get the number of parts in the message
INT nPartCount = MimeGetMessagePartCount(hMessage);

for (INT nMessagePart = 0; nMessagePart < nPartCount; nMessagePart++)
{
    // Set the current message part
    MimeSetMessagePart(hMessage, nMessagePart);

    if (MimeIsContentType(hMessage, MIME_CURRENT_PART, _T("text/html")))
    {
        DWORD dwOptions = MIME_FILTER_STRICT;

        // Determine the amount of memory we need to allocate
        LPTSTR pszFiltered = NULL;
        LONG cchFiltered = MimeFilterMessageText(hMessage, NULL, 0, dwOptions);

        if (cchFiltered < 1)
        {
            // Unable to filter this message part even though its identified
            // as containing HTML or it does not contain any text
            break;
        }
```

```
        else
        {
            // Allocate memory for the filtered text; note the extra space is
            // included for the terminating null character
            pszFiltered = (LPTSTR)malloc((cchFiltered + 1) * sizeof(TCHAR));

            if (pszFiltered == NULL)
            {
                // Unable to allocate memory for the filtered text
                break;
            }

            // Return the filtered message contents in pszFiltered
            cchFiltered = MimeFilterMessageText(
                                hMessage,
                                pszFiltered,
                                cchFiltered + 1,
                                dwOptions);

            if (cchFiltered > 0)
            {
                // Write the filtered text to standard output on the console
                fwrite(pszFiltered, sizeof(TCHAR), cchFiltered, stdout);
            }

            // Free the memory we allocated for the filtered text
            free(pszFiltered);
            break;
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetContentType, MimeIsContentType, MimeCompareMessageText, MimeGetMessageText, MimeNormalizeText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeFindAttachment Function

```
INT WINAPI MimeFindAttachment(
   HMESSAGE hMessage,
   LPCTSTR lpszFileName
);
```

The **MimeFindAttachment** function searches the message for an attachment with the specified file name.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string that specifies the name of the file attachment to search for. This parameter should only specify a base file name; it should not include a file path and cannot be NULL.

Return Value

If the function succeeds, the return value is the message part number that contains an attachment that matches the specified file name. If the message does not contain an attachment with the specified file name, the function will return MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function will search the message for a attachment that matches the specified file name. The search is not case-sensitive, however it must match the attachment file name completely. This function will not match partial file names or names that include wildcard characters. To obtain a list of all of the files attached to a message, use the **MimeEnumAttachments** function.

Example

```
// The name of the file attachment to search for
LPCTSTR lpszFileName = _T("MyProject.docx");

// Search for the attached file and store it on the local system
INT nMessagePart = MimeFindAttachment(hMessage, lpszFileName);
if (nMessagePart != MIME_ERROR)
{
    MimeSetMessagePart(hMessage, nMessagePart);

    if (MimeExtractFile(hMessage, lpszFileName) != MIME_ERROR)
        _tprintf(_T("Saved file attachment %s\n"), lpszFileName);
    else
    {
        _tprintf(_T("Unable to save file attachment %s\n"), lpszFileName);
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeExtractAllFiles, MimeExtractFile, MimeExtractFileEx, MimeGetAttachedFileName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeFindStoredMessage Function

```
LONG WINAPI MimeFindStoredMessage(
   HMESSAGESTORE hStorage,
   LONG nMessageId,
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue,
   DWORD dwOptions
);
```

The **MimeFindStoredMessage** function searches for a message in the specified message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which specifies the message number that should be used when starting the search. The first message in the message store has a value of one.

lpszHeaderName

The string which specifies the name of the header field that should be searched. The header field name is not case sensitive. This parameter cannot be NULL.

lpszHeaderValue

The string which specifies the header value that should be searched for. This may be a literal string or it may specify a regular expression. The search options can be used to specify if the search is case-sensitive, and whether the search should return partial matches to the string. This parameter cannot be NULL.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_SEARCH_DEFAULT	Perform a complete match against the specified header value. The comparison is not case-sensitive.
MIME_SEARCH_CASE_SENSITIVE	The header field value comparison will be case-sensitive. Note that this does not affect header field names. Matches for header names are always case-insensitive.
MIME_SEARCH_PARTIAL_MATCH	Perform a partial match against the specified header value. It recommended that this option be used when searching for matches to email addresses.
MIME_SEARCH_DECODE_HEADERS	Decode any encoded message headers before comparing them to the specified value.
MIME_SEARCH_ANCHORED_MATCH	Perform a complete match against the specified header value. This is the default behavior if a partial match is not specified.

MIME_SEARCH_LITERAL_MATCH	Perform a literal match against the specified header value, even if it may include a regular expression.

Return Value

If the function succeeds, the return value is the number for the message which matches the search criteria. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeFindStoredMessage** function is used to search the message store for a message which matches a specific header field value. For example, it can be used to find every message which is addressed to a specific recipient or has a subject which matches a particular string value.

If the *lpszHeaderValue* parameter does not contain any regular expression constructs (such as **^**, **$**, **[ ]**, or quantifiers like ***** and **+**), the search is performed using a simple substring comparison. If a regular expression pattern is specified, the function compiles the pattern and searches the message headers for a match. The regular expression matching uses a simplified version which only supports anchors, wildcards, match classes (enclosed in brackets) and a limited number of escape sequences such as **\d** to match any digit or **\s** to match any space character. It does not support grouping or replacement.

If you want to search for a message where the header value may contain characters typically used in regular expressions such as **[** or **$**, you can either escape them using a backslash, or you can use the MIME_SEARCH_LITERAL_MATCH option which forces the function to only perform a substring search.

Example

```
HMESSAGE hMessage = INVALID_MESSAGE;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
LONG nMessageId = 1;

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = MimeFindStoredMessage(hStorage,
                                       nMessageId,
                                       lpszHeader,
                                       lpszAddress,
                                       MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        // Get a handle to the message that was found
        hMessage = MimeGetStoredMessage(hStorage, nMessageId, 0);

        if (hMessage != INVALID_MESSAGE)
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = MimeExportMessage(hMessage, szFileName);
```

```
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetStoredMessage, MimeGetStoredMessageCount, MimeDeleteStoredMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeFormatDate Function

```
LPTSTR WINAPI MimeFormatDate(
    LONG nSeconds,
    LONG nTimezone,
    LPTSTR lpszDate,
    INT cchMaxDate
);
```

The **MimeFormatDate** converts the specified date, expressed as the number of seconds since 1 January 1970, into a string compatible with the RFC 822 standard format for email messages.

Parameters

nSeconds

An integer which specifies the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C/C++ time functions. If the value of this parameter is zero, the current date and time is used.

nTimezone

An integer which is set to the difference in seconds between the specified date's timezone and Coordinated Universal Time. A value of zero specifies Coordinated Universal Time (UTC), while a positive value specifies a timezone west of UTC and a negative value specifies a timezone east of UTC. For example, Eastern Standard Time would be specified as 18000 and Pacific Standard Time would be 28800.

lpszDate

A null-terminated string buffer which will contain the formatted date. This parameter cannot be a NULL pointer.

cchMaxDate

The maximum number of characters, including the terminating null character, which may be copied into the date string buffer.

Return Value

If the function succeeds, a pointer to the date string buffer is returned. If the function fails, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Remarks

The date string is returned in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. This format is as follows:

```
www, dd mmm yyyy hh:mm:ss [-]zzzz
```

Each part of the date string is defined as follows:

Format	Description
www	Weekday
dd	Day
mmm	Month
yyyy	Year
hh	Hour (24-hour clock)

mm	Minutes
ss	Seconds
zzzz	Timezone

The weekday and month are displayed using standard three-character English abbreviations. The timezone is displayed as the difference (in hours and minutes) between the specified timezone and Coordinated Universal Time. For example, if the timezone is eight hours west of Coordinated Universal Time, the ***nTimezone*** value would be 28800. This would be displayed as -0800 in the formatted date string.

Note that the format of the date string is defined by the RFC 822 standard, and is not affected by localization settings on the host system.

Developers are strongly encouraged to use the **MimeFormatDate64** function for all new development. This function represents date and time values as 32-bit integers, which limits the representable range to dates before 19 January 2038 (the so-called "Year 2038 problem"). After this date, 32-bit time values will overflow, resulting in incorrect date calculations and potential failures in software that processes email message dates. By contrast, the 64-bit versions use 64-bit integer values, matching the standard **`time_t`** type used by modern C/C++ libraries. This not only eliminates the Year 2038 limitation but also ensures compatibility with current and future systems, allowing correct handling of dates both far in the past and well into the future. Whenever possible, applications should transition to these 64-bit functions to ensure long-term reliability and portability.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFormatDate64, MimeParseDate, MimeParseDate64, MimeGetMessageDate, MimeSetMessageDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeFormatDate64 Function

```
LPTSTR WINAPI MimeFormatDate64(
    LONGLONG nSeconds,
    LONG nTimezone,
    LPTSTR lpszDate,
    INT nMaxLength
);
```

The **MimeFormatDate** converts the specified date, expressed as the number of seconds since 1 January 1970, into a string compatible with the RFC 822 standard format for email messages.

Parameters

nSeconds

A 64-bit integer which specifies the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C/C++ time functions. If the value of this parameter is zero, the current date and time is used.

nTimezone

A 32-bit integer which is set to the difference in seconds between the specified date's timezone and Coordinated Universal Time. A value of zero specifies Coordinated Universal Time (UTC), while a positive value specifies a timezone west of UTC and a negative value specifies a timezone east of UTC. For example, Eastern Standard Time would be specified as 18000 and Pacific Standard Time would be 28800.

lpszDate

A null-terminated string buffer which will contain the formatted date. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters, including the terminating null character, which may be copied into the date string buffer.

Return Value

If the function succeeds, a pointer to the date string buffer is returned. If the function fails, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Remarks

The date string is returned in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. This format is as follows:

```
www, dd mmm yyyy hh:mm:ss [-]zzzz
```

Each part of the date string is defined as follows:

Format	Description
www	Weekday
dd	Day
mmm	Month
yyyy	Year
hh	Hour (24-hour clock)

mm	Minutes
ss	Seconds
zzzz	Timezone

The weekday and month are displayed using standard three-character English abbreviations. The timezone is displayed as the difference (in hours and minutes) between the specified timezone and Coordinated Universal Time. For example, if the timezone is eight hours west of Coordinated Universal Time, the ***nTimezone*** value would be 28800. This would be displayed as -0800 in the formatted date string.

Note that the format of the date string is defined by the RFC 822 standard, and is not affected by localization settings on the host system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeParseDate64, MimeGetMessageDate, MimeSetMessageDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetAllHeaders Function

```
INT WINAPI MimeGetAllHeaders(
   HMESSAGE hMessage,
   LPTSTR lpszHeaders,
   INT nMaxLength
);
```

The **MimeGetAllHeaders** function returns the complete RFC 822 header values in a string buffer.

Parameters

hMessage

Handle to the message.

lpszHeaders

Pointer to string buffer which will contain the header values for the specified message. This parameter may be NULL, in which case the function will calculate the number of characters needed to store the complete header block.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the *lpszHeaders* string. If the *lpszHeaders* parameter is NULL, this value must be zero. If the *lpszHeaders* parameter is not NULL, this value must be large enough to store the entire list of addresses.

Return Value

If the function succeeds and the *lpszHeaders* parameter is NULL, the return value is the minimum number of characters that should be allocated to store all of the header values, including the terminating null character. If the *lpszHeaders* parameter is not NULL, then the return value is the number of characters copied into the string, not including the terminating null character. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetAllHeaders** function will return all of the RFC 822 header values in a string buffer. This includes the message headers that are most commonly referred to, such as the To, From and Subject headers. Each header and its value are separated by a colon, and terminated with a carriage return and linefeed (CRLF) pair.

The headers and their values returned by this function will not be identical to the header block in the original message. If a header value is split across multiple lines, this function will fold the text, returning the complete header value on a single line of text and removing any extraneous whitespace. If the header value has been encoded by the mail client, this function will return the decoded value, not the original encoded value.

Example

```
LPTSTR lpszHeaders = NULL;
INT nLength;

// Determine the number of characters that should be allocated to store
// the RFC822 headers
```

```
nLength = MimeGetAllHeaders(hMessage, NULL, 0);

// Allocate the memory for the string buffer that is large enough and
// call MimeGetAllHeaders again

if (nLength > 0)
{
    lpszHeaders = (LPTSTR)LocalAlloc(LPTR, nLength * sizeof(TCHAR));
    if (lpszHeaders == NULL)
        return; // Virtual memory exhausted

    nLength = MimeGetAllHeaders(hMessage, lpszHeaders, nLength);
}

// The lpszHeaders string now contains all of the RFC822 headers for the
// message, with each header terminated by a CRLF sequence

if (lpszHeaders != NULL)
{
    LocalFree((HLOCAL)lpszHeaders);
    lpszHeaders = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetNextMessageHeader, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetAllRecipients Function

```
INT WINAPI MimeGetAllRecipients(
   HMESSAGE hMessage,
   LPCTSTR lpszExtraAddress,
   LPTSTR lpszRecipients,
   INT nMaxLength
);
```

The **MimeGetAllRecipients** function returns a comma-separated list of recipient addresses in a string buffer.

Parameters

hMessage

Handle to the message.

lpszExtraAddress

A null-terminated string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpszRecipients

Pointer to string buffer which will contain a comma-separated list of email addresses when the function returns. This parameter may be NULL, in which case the function will calculate the number of characters needed to store the complete list.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the ***lpszRecipients*** string. If the ***lpszRecipients*** parameter is NULL, this value must be zero. If the ***lpszRecipients*** parameter is not NULL, this value must be large enough to store the entire list of addresses.

Return Value

If the function succeeds and the ***lpszRecipients*** parameter is NULL, the return value is the minimum number of characters that should be allocated to store the list recipient addresses, including the terminating null character. If the ***lpszRecipients*** parameter is not NULL, then the return value is the number of characters copied into the buffer, not including the terminating null character. If the function returns a value of zero, then the specified message has no recipients. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetAllRecipients** function is useful for creating a list of message recipients that can be passed to functions like **SmtpSendMessage**. If you wish to dynamically allocate the string buffer that will contain the list of recipients, then the ***lpszRecipients*** parameter should be NULL and the ***nMaxLength*** parameter should have a value of zero. The function will then return the recommended size of the buffer that should be allocated. This value is guaranteed to be large enough to store the entire list of message recipients, including the terminating null character.

Example

```
LPTSTR lpszRecipients = NULL;
INT nLength;
```

```
// Determine the number of characters that should be allocated to store
// a list of the recipient addresses in the message

nLength = MimeGetAllRecipients(hMessage, NULL, NULL, 0);

// Allocate the memory for the string buffer that is large enough and
// call MimeGetAllRecipients again

if (nLength > 0)
{
    lpszRecipients = (LPTSTR)LocalAlloc(LPTR, nLength * sizeof(TCHAR));
    if (lpRecipients == NULL)
        return; // Virtual memory exhausted

    nLength = MimeGetAllRecipients(hMessage, NULL, lpszRecipients, nLength);
}

// The lpszRecipients string now contains a comma-separated list of
// each recipient address in the message

if (lpszRecipients != NULL)
{
    LocalFree((HLOCAL)lpszRecipients);
    lpszRecipients = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageRecipients, MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetNextMessageHeader, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetAttachedFileName Function

```
BOOL WINAPI MimeGetAttachedFileName(
    HMESSAGE hMessage,
    LPTSTR lpszFileName,
    INT cchFileName
);
```

The **MimeGetAttachedFileName** function returns the file name for the attachment to the current message part.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a buffer that will contain the current attachment file name as a string.

cchFileName

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The function will first try to get the filename from the Content-Disposition header field. If this field does not exist, it then attempts to get the name from the Content-Type header field. If neither field exists, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeEnumAttachments, MimeExtractFile, MimeFindAttachment

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetAuthenticationResults Function

```
DWORD WINAPI MimeGetAuthenticationResults(
    HMESSAGE hMessage
);
```

The **MimeGetAuthenticationResults** function returns information about the authentication status of the message.

Parameters

hMessage

A handle to the message.

Return Value

This function returns one or more flags based on the value of the `Authentication-Results` header in the message. This function explicitly clears the last error code to assist in differentiating between an error condidition, such as an invalid message handle, and when no authentication information was available. The following flags are defined:

Value	Description
MIME_AUTH_NONE	The message does not contain an Authentication-Results header or does not include any recognizable authentication types and their associated results.
MIME_AUTH_SPF_PASSED	The message passed the Sender Policy Framework (SPF) check. The sender's IP address was authorized to send mail for the domain in the envelope sender address.
MIME_AUTH_SPF_FAILED	The message failed the SPF check. The sending mail server is not authorized by the domain's SPF record to send mail on its behalf.
MIME_AUTH_DKIM_PASSED	The message passed the DomainKeys Identified Mail (DKIM) check. The signature applied by the sender was successfully verified using the domain's public key.
MIME_AUTH_DKIM_FAILED	The message failed the DKIM check. The signature could not be verified or did not match the message content, suggesting it may have been altered.
MIME_AUTH_DMARC_PASSED	The message passed DMARC evaluation. The message aligned with a valid SPF or DKIM result based on the sender domain's policy.
MIME_AUTH_DMARC_FAILED	The message failed DMARC evaluation. It did not pass SPF or DKIM checks in a way that aligns with the domain's published policy.
MIME_AUTH_TEMP_ERROR	A temporary error occurred while evaluating one or more authentication mechanisms. This may indicate a DNS lookup timeout or other transient failure.
MIME_AUTH_PERM_ERROR	A permanent error occurred during authentication checks. This may indicate a malformed DNS record or invalid

	message structure that prevents proper evaluation.

Remarks

Modern email systems use a set of authentication mechanisms to help verify that a message was sent from a legitimate source and not forged by a spammer or attacker. The **MimeGetAuthenticationResults** function enables you to easily determine which of these mechanisms have passed a check and which of them have failed. It evaluates the following authentication information reported by the mail server:

SPF
Sender Policy Framework allows the owner of a domain to specify which mail servers are authorized to send email on its behalf. If the sending server is not listed in the domain's SPF record, the message may be marked as suspicious. SPF helps prevent spoofing of the envelope sender address, which is often used in phishing attacks.

DKIM
DomainKeys Identified Mail uses public-key cryptography to sign specific parts of the email, such as headers and body content. The signature is added by the sending mail server, and receiving systems use DNS to retrieve the sender's public key to verify that the message has not been tampered with. A valid DKIM signature helps confirm that the message was truly sent by the claimed domain.

DMARC
Domain-based Message Authentication, Reporting, and Conformance builds on SPF and DKIM by defining a policy for how a domain wants email receivers to handle messages that fail authentication checks. It also ensures that the visible From: address matches the domain that passed SPF or DKIM (a concept called alignment). DMARC allows domain owners to monitor abuse and enforce stronger protection against spoofing.

Checking these authentication results can help identify spoofed or forged messages, improve filtering decisions, and increase overall mail security. While not every legitimate message will have all three, failure of all mechanisms, especially from domains that publish strict DMARC policies, should be treated as suspicious.

While these authentication methods are useful signals, failing one of them does not necessarily mean a message is malicious. Many legitimate messages, even from well-known companies, may have misconfigured SPF records, missing DKIM signatures, or inconsistent DMARC policies. Conversely, a message that passes all checks could still be unwanted or deceptive. These results should be treated as advisory signals rather than absolute indicators of legitimacy. Developers are encouraged to consider authentication status alongside other factors, such as message content, sender reputation, and user reporting.

To simplify checking authentication outcomes, the following predefined flag combinations are also available:

- MIME_AUTH_PASSED is a combination of all passing results (SPF, DKIM, DMARC).
- MIME_AUTH_FAILED is a combination of all failing results.
- MIME_AUTH_ERROR is a combination of temporary or permanent errors.

For a simplified boolean check using these flags, see the **MimeCheckAuthenticationResults** function.

Example

```
HMESSAGE hMessage = MimeOpenMessage(lpszFileName, MIME_OPEN_DEFAULT);

if (hMessage != INVALID_MESSAGE)
{
    // Get the authentication results for the message
    DWORD dwAuthFlags = MimeGetAuthenticationResults(hMessage);

    // Check if any of the SPF, DKIM, or DMARC checks failed
    if (dwAuthFlags & MIME_AUTH_FAILED)
        _tprintf(_T("This message has failed one or more authentication
checks\n"));

    MimeCloseMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCheckAuthenticationResults, MimeGetMessageHeader, MimeGetMessageHeaderEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetContentDigest Function

```
INT WINAPI MimeGetContentDigest(
    HMESSAGE hMessage,
    LPTSTR lpszDigest,
    INT cchDigest
);
```

The **MimeGetContentDigest** function returns an encoded digest of the message.

Parameters

hMessage

Message handle.

lpszDigest

Pointer to a string buffer to contain the MD5 digest for the specified message.

cchDigest

Maximum length of the digest string, in bytes.

Return Value

If the function succeeds, the return value is the length of the message digest string. A value of zero specifies that there is no MD5 digest for the current message. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function returns the value of the Content-MD5 header field in the main body of the message. If the header exists, it contains the MD5 digest for the message as defined in RFC 1864.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetContentLength Function

```
LONG WINAPI MimeGetContentLength(
    HMESSAGE hMessage
);
```

The **MimeGetContentLength** function returns the size of the current message part content in bytes.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the content length. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function will return the size of the content in the current message part as the number of bytes and does not account for any Unicode conversion of text. Exercise caution when using this function to determine the size of the buffer that should be allocated for a function like **MimeGetMessageText**. You should always allocate enough memory to accommodate any potential text conversion and decoding which may occur.

It is important to note that this function does not return the number of characters of text in the message. The number of characters and bytes would only be the same when working with messages which contain plain, unencoded ASCII text. Messages which are encoded and/or multibyte character sets can represent a single character with multiple byte sequences. In this case, the content length will not accurately represent the number of characters copied to a string buffer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeFilterMessageText, MimeGetMessageHeader, MimeGetMessagePart, MimeGetMessageText, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetContentType Function

```
INT WINAPI MimeGetContentType(
    HMESSAGE hMessage
);
```

Parameters

hMessage

Handle to the message whose content type should be retrieved. The current message part is used.

Return Value

The function returns an integer value that identifies the top-level media type for the current message part. This value may be one of the following predefined constants:

Value	Description
MIME_CONTENT_TEXT	Textual content (e.g.: text/plain, text/html)
MIME_CONTENT_IMAGE	Image content (e.g.: image/jpeg, image/png)
MIME_CONTENT_AUDIO	Audio content (e.g.: audio/mpeg)
MIME_CONTENT_VIDEO	Video content (e.g.: video/mp4)
MIME_CONTENT_APPLICATION	Application-specific data (e.g.: application/pdf)
MIME_CONTENT_MESSAGE	Embedded message (e.g.: message/rfc822)
MIME_CONTENT_MULTIPART	Multipart content (e.g.: multipart/alternative)
MIME_CONTENT_WIDETEXT	Extended text using wide character encoding
MIME_CONTENT_UNKNOWN	The content type is unknown or could not be determined

If the function fails, it returns **MIME_ERROR**. To retrieve extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetContentType** function parses the `Content-Type` header of the current message part and returns the content type in a normalized form. This function eliminates the need for manual parsing of MIME headers and is useful for determining how message parts should be processed based on their media type. For example, consider a current message part with:

```
Content-Type: text/html; charset=utf-8
```

In this case, the function will return `MIME_CONTENT_TEXT`. This function always returns the content type for the current message part set by the **MimeSetMessagePart** function.

Example

```
if (MimeGetContentType(hMessage) == MIME_CONTENT_TEXT)
{
    // Get the size of the decoded/converted message text
    LONG cchMessage = MimeGetMessageText(hMessage, 0, NULL, 0);
    if (cchMessage < 1)
    {
        _tprintf(_T("Message part %d does not contain any text\n"), nMessagePart);
```

```
        return;
    }

    // Allocate a buffer for the message text on the heap
    LPTSTR pszMessage = (LPTSTR)malloc((cchMessage + 1) * sizeof(TCHAR));
    if (pszMessage == NULL) // Virtual memory exhausted
        return;

    // Copy the message contents to the buffer
    cchMessage = MimeGetMessageText(hMessage, 0, pszMessage, cchMessage + 1);
    if (cchMessage > 0)
    {
        // Write the contents of the message part to the console
        fwrite(pszMessage, sizeof(TCHAR), cchMessage, stdout);
        fflush(stdout);
    }

    // Free the memory allocated for our text buffer
    free(pszMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeIsContentType, MimeGetMessageHeader, MimeGetMessageText, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetContentTypeEx Function

```
INT WINAPI MimeGetContentTypeEx(
    HMESSAGE hMessage,
    INT nMessagePart,
    LPTSTR lpszSubType,
    INT cchSubType,
    LPTSTR lpszParams,
    INT cchParams
);
```

Parameters

hMessage

Handle to the message whose content type should be retrieved.

nMessagePart

An integer value which specifies the message part. Message parts are zero-based with the part zero being the primary content and additional parts typically containing message content and file attachments. A value of MIME_CURRENT_PART (-1) specifies the content type for the current message part should be returned.

lpszSubType

A null-terminated string buffer that will receive the content subtype, such as "html" or "plain". This parameter may be NULL if the subtype is not required.

cchSubType

The maximum number of characters that can be copied into the ***lpszSubType*** buffer, including the terminating null character. If the buffer is too small, the function will return an error.

lpszParams

A null-terminated string buffer that will receive any parameters associated with the content type, such as `charset=utf-8`. This parameter may be NULL if parameters are not needed.

cchParams

The maximum number of characters that can be copied into the ***lpszParams*** buffer, including the terminating null character. If the buffer is too small, the function will return an error.

Return Value

The function returns an integer value that identifies the top-level media type for the current message part. This value may be one of the following predefined constants:

Value	Description
MIME_CONTENT_TEXT	Textual content (e.g.: text/plain, text/html)
MIME_CONTENT_IMAGE	Image content (e.g.: image/jpeg, image/png)
MIME_CONTENT_AUDIO	Audio content (e.g.: audio/mpeg)
MIME_CONTENT_VIDEO	Video content (e.g.: video/mp4)
MIME_CONTENT_APPLICATION	Application-specific data (e.g.: application/pdf)
MIME_CONTENT_MESSAGE	Embedded message (e.g.: message/rfc822)
MIME_CONTENT_MULTIPART	Multipart content (e.g.: multipart/alternative)

MIME_CONTENT_WIDETEXT	Extended text using wide character encoding
MIME_CONTENT_UNKNOWN	The content type is unknown or could not be determined

If the function fails, it returns MIME_ERROR. To retrieve extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetContentTypeEx** function parses the `Content-Type` header of the current message part and returns the content type information in a normalized form. If requested, the subtype name and any optional parameters are copied to the caller-supplied buffers. This function eliminates the need for manual parsing of MIME headers and is useful for determining how message parts should be processed based on their media type. For example, consider a message part with:

```
Content-Type: text/html; charset=utf-8
```

This function will return `MIME_CONTENT_TEXT`, a subtype of `"html"`, and the parameter string `"charset=utf-8"`. If only the numeric media type is required for use in a `switch` or `if-then-else` statement, the content type and parameters buffers may be passed as `NULL`.

If you wish to determine if the content type for a message part matches a specific value, you can use the **MimeIsContentType** function.

Example

```
TCHAR szSubType[32] = {0};
TCHAR szParams[64] = {0};

// Check if the current message part is plain text; szParams may
// contain the original character set used
INT nContentType = MimeGetContentTypeEx(
                        hMessage,
                        MIME_CURRENT_PART,
                        szSubType, _countof(szSubType),
                        szParams, _countof(szParams));

if (nContentType == MIME_CONTENT_TEXT && lstrcmpiA(szSubType, "plain") == 0)
{
    // Get the size of the decoded/converted message text
    LONG cchMessage = MimeGetMessageText(hMessage, 0, NULL, 0);
    if (cchMessage < 1)
    {
        _tprintf(_T("Message part %d does not contain any text\n"), nMessagePart);
        return;
    }

    // Allocate a buffer for the message text on the heap
    LPTSTR pszMessage = (LPTSTR)malloc((cchMessage + 1) * sizeof(TCHAR));
    if (pszMessage == NULL) // Virtual memory exhausted
        return;

    // Copy the message contents to the buffer
    cchMessage = MimeGetMessageText(hMessage, 0, pszMessage, cchMessage + 1);
    if (cchMessage > 0)
    {
        // Write the contents of the message part to the console
        fwrite(pszMessage, sizeof(TCHAR), cchMessage, stdout);
```

```
        fflush(stdout);
    }

    // Free the memory allocated for our text buffer
    free(pszMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetContentType, MimeIsContentType, MimeGetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetErrorString Function

```
INT WINAPI MimeGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **MimeGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **MimeGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetLastError, MimeSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetExportOptions Function

```
BOOL WINAPI MimeGetExportOptions(
    HMESSAGE hMessage
    LPDWORD lpdwOptions
);
```

The **MimeGetExportOptions** function returns a bitmask that describes current message export options.

Parameters

hMessage

Handle to the message.

lpdwOptions

Pointer to mask of attribute options. The mask is a combination of the following values:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is 0. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csmsgv12.lib

See Also

MimeExportMessage, MimeImportMessage, MimeSetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetFileContentType Function

```
UINT WINAPI MimeGetFileContentType(
   HMESSAGE hMessage,
   LPCTSTR lpszFileName,
   LPTSTR lpszContentType,
   INT cchContentType
);
```

The **MimeGetFileContentType** function returns the content type for the specified file.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the file for which content type information is returned.

lpszContentType

Pointer to a string buffer that will contain the MIME type for the specified file. This may be a NULL pointer, in which case this parameter is ignored. If a buffer is provided, it is recommended that it be at least 64 characters in length.

cchContentType

An integer which specifies the maximum number of characters, including the terminating null character, which may be copied into the string buffer.

Return Value

If the function succeeds, the return value is the content type of the specified file. If the function fails, the return value is MIME_CONTENT_UNKNOWN. To get extended error information, call **MimeGetLastError**.

The following values may be returned by this function:

Value	Description
MIME_CONTENT_UNKNOWN	The file content type is unknown. This value may be returned if the message handle is invalid, or if the file extension is unknown and the file could not be opened for read access.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only

	printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

Remarks

The content type for a given file is determined based on the file name extension, or if the extension is not recognized, the actual contents of the file. On 32-bit platforms, the system registry is used to determine the default content type values for a given extension. In all cases, file types that are explicitly set using the **MimeSetFileContentType** function will override the default system values.

The content type string which is copied to the string buffer is the standard MIME content type description, which specifies a primary type and a subtype, separated by a slash. For example, a plain text file would have a content type of text/plain, while an HTML document would have a content type of text/html. Binary files may be associated with a specific application. For example, the content type for a Microsoft Word document is application/msword. Those binary files which are not associated with a specific application, or have an unrecognized file name extension, have a content type of application/octet-stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachData, MimeAttachFile, MimeSetFileContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetFirstMessageHeader Function

```
BOOL WINAPI MimeGetFirstMessageHeader(
    HMESSAGE hMessage,
    LPTSTR lpszHeader,
    INT cchMaxHeader,
    LPTSTR lpszValue,
    INT cchMaxValue
);
```

The **MimeGetFirstMessageHeader** function returns the header field name and value for the first header in the current message part. This function is typically used in conjunction with **MimeGetNextMessageHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

hMessage

Handle to the message.

lpszHeader

A null-terminated string buffer that will contain the name of the first header in the current message part. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A null-terminated string buffer that will contain the value of the first header in the current message part. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is non-zero. If no headers exist for the current message part, or the handle to the message is invalid, the function will return zero. To get extended error information, call **MimeGetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **MimeSetMessagePart** function.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageHeaders, MimeGetMessagePart, MimeGetNextMessageHeader, MimeParseHeader, MimeSetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetLastError Function

```
DWORD WINAPI MimeGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last error code value. Functions set this value by calling the **MimeSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

You should call the **MimeGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **MimeSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those functions which call **MimeSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeGetErrorString, MimeSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageBoundary Function

```
LPCTSTR WINAPI MimeGetMessageBoundary(
    HMESSAGE hMessage
);
```

The **MimeGetMessageBoundary** function returns a pointer to the boundary string used to separate the parts of a multipart message.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is a pointer to the boundary string. If the function fails, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCreateMessagePart, MimeGetMessageHeader, MimeGetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageDate Function

```
LPCTSTR WINAPI MimeGetMessageDate(
    HMESSAGE hMessage,
    BOOL bLocalize
);
```

The **MimeGetMessageDate** function returns a pointer to a string that contains the message date and time.

Parameters

hMessage

Handle to the message.

bLocalize

Boolean flag which specifies if the date and time should be localized for the current timezone.

Return Value

If the function succeeds, the return value is a pointer to the date and time string. If the function fails, it will return a NULL pointer. To get extended error information, call **MimeGetLastError**.

Remarks

If no date has been specified in the message, the Date header field will be set to the current date and time, and that value will be returned. The date string returned by this function should never be directly modified by the application. Each call to this function will invalidate the previous value that was returned, so if you wish to save or modify the value, you should first make a private copy of the string.

To convert this date string to a long integer value that can be used with the standard C/C++ time functions, use the **MimeParseDate64** function. Refer to the **MimeFormatDate64** function for information on the format of the date string.

Developers are encouraged to use the **MimeGetMessageDateEx** function for all new development. This function returns a pointer to a string managed by the internal message context, which may become invalid or be overwritten by subsequent operations. In addition, if the message does not contain a `Date` header, this function will automatically generate and insert a current date value, modifying the message content as a side effect. By contrast, **MimeGetMessageDateEx** copies the date string directly into a caller-supplied buffer and does not alter the message if the header is missing. This results in more predictable and thread-safe behavior, with no risk of unintentional modifications to the message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFormatDate, MimeFormatDate64, MimeGetMessageDateEx, MimeGetMessageHeader, MimeParseDate, MimeParseDate64, MimeSetMessageDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageDateEx Function

```
INT WINAPI MimeGetMessageDateEx(
    HMESSAGE hMessage,
    LPTSTR lpszDate,
    INT nMaxLength,
    BOOL bLocalize
);
```

The **MimeGetMessageDateEx** function copies the current message date to the specified string buffer.

Parameters

hMessage

Handle to the message.

lpszDate

A null-terminated string buffer which will contain the message date. This parameter cannot be NULL.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. This parameter cannot be zero.

bLocalize

A boolean flag which specifies if the date and time should be localized for the current time zone.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer., not including the terminating null character A return value of zero indicates the message does not have a valid `Date` header or an error has occurred. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetMessageDateEx** function copies the formatted date string directly into the buffer provided by the caller. This ensures the date string remains valid for the lifetime of the caller's buffer, avoiding issues with internal pointer reuse or invalidation that could occur in multithreaded or reentrant scenarios. If the message contains a valid `Date` or `Sent` header, the function retrieves and, if requested, localizes the date and time string before copying it to the specified buffer. The returned string always conforms to the standard format described in RFC 822. To convert this date string into a numeric value suitable for time calculations, use the **MimeParseDate64** function.

A key difference between this function and the behavior of the **MimeGetMessageDate** function is that this function will not automatically create or set a `Date` header if it is missing from the message. If neither a `Date` nor `Sent` header is present, the function simply fails and returns zero, rather than inserting the current date and time into the message. This change eliminates side effects and ensures that calling this function does not modify the underlying message content.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFormatDate64, MimeGetMessageDate, MimeGetMessageHeader, MimeParseDate64, MimeSetMessageDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageHeader Function

```
LPCTSTR WINAPI MimeGetMessageHeader(
    HMESSAGE hMessage,
    LPCTSTR lpszHeader
);
```

The **MimeGetMessageHeader** function returns a pointer to the value associated with the specified header field in the current message part.

Parameters

hMessage

Handle to the message.

lpszHeader

A null-terminated string which specifies the message header. This parameter cannot be a NULL pointer.

Return Value

If the function succeeds, the return value is a pointer to the header value. If the header field does not exist, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **MimeSetMessagePart** function. For languages other than C/C++, it may be preferable to use the **MimeGetMessageHeaderEx** function which copies the header value into a string buffer rather than returning a pointer to the header value itself.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageHeaders, MimeGetFirstMessageHeader, MimeGetMessageHeaderEx, MimeGetMessagePart, MimeGetNextMessageHeader, MimeParseHeader, MimeSetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageHeaderEx Function

```
INT WINAPI MimeGetMessageHeaderEx(
    HMESSAGE hMessage,
    INT nMessagePart,
    LPCTSTR lpszHeader,
    LPTSTR lpszValue,
    INT nMaxValue,
    DWORD dwReserved
);
```

The **MimeGetMessageHeaderEx** copies the value of the specified header into a string buffer.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value which specifies which part of the message to return the header value from. A value of zero returns a header value from the main message header, while a value greater than zero returns the header value from that specific part of a multipart message. A value of -1 specifies that the header value should be returned from the current message part.

lpszHeader

A null-terminated string which specifies the message header. This parameter cannot be a NULL pointer.

lpszValue

A null-terminated string buffer which will contain the value of the specified header when the function returns. If this parameter is NULL, the function will return the length of the header value without copying the data. This is useful for determining the length of a header value so that a string buffer can be allocated and passed to a subsequent call to the function.

nMaxValue

An integer value which specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this value should be zero.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null byte. If the ***lpszValue*** parameter is NULL, the return value is the length of the header value. If the header field does not exist, a value of zero is returned. If an invalid pointer or message part is specified, a value of MIME_ERROR is returned. To get extended error information, call **MimeGetLastError**.

Example

```
LPTSTR lpszValue = NULL;
INT cchValue = 0;

// Determine the length of the To header field
```

```
cchValue = MimeGetMessageHeaderEx(hMessage, 0, _T("To"), NULL, 0);

// If the header field exists, allocate a string buffer
// and copy the value into the buffer
if (cchValue > 0)
{
    lpszValue = (LPTSTR)LocalAlloc(LPTR, cchValue + 1);
    MimeGetMessageHeaderEx(hMessage,
                           0,
                           _T("To"),
                           lpszValue,
                           cchValue + 1);
}

// After the string buffer has been used, release the
// memory that was allocated for it
if (lpszValue)
{
    LocalFree((HLOCAL)lpszValue);
    lpszValue = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageHeaders, MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetMessagePart, MimeGetNextMessageHeader, MimeParseHeader, MimeSetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageLinks Function

```
LONG WINAPI MimeGetMessageLinks(
   HMESSAGE hMessage,
   DWORD dwLinkTypes,
   LPTSTR lpszLinkList,
   LONG nMaxLength,
   LONG *lpnLinkCount
);
```

The **MimeGetMessageLinks** function returns the hyperlinks in the current message part.

Parameters

hMessage

Handle to the message.

dwLinkTypes

An unsigned integer value which specifies one or more bitflags which identify what types of links you would like to extract. To extract all supported link types, use a value of zero.

Value	Description
MIME_LINK_HTTP	Return standard **http://** links in the current message part.
MIME_LINK_HTTPS	Return secure **https://** links in the current message part.
MIME_LINK_FTP	Return standard **ftp://** links in the current message part.
MIME_LINK_FTPS	Return secure **ftps://** and **ftpes://** links in the current message part.
MIME_LINK_SFTP	Return secure **sftp://** links in the current message part.
MIME_LINK_FILE	Return local **file://** links in the current message part.

lpszLinkList

A null-terminated string buffer which will contain all of the matching links found in the current message part. This buffer must be large enough to contain the entire list, including the terminating null character. If you wish to determine the number of characters required to store the list, this parameter can be NULL and the ***nMaxLength*** parameter should be zero. If you are dynamically allocating memory for the list, make sure you are including additional space for the terminating null. If multiple links are found in the message contents, each link will be separated with a single space. This makes the list easy to parse using basic string functions such as the **strtok** or **Split** methods in most languages.

nMaxLength

The maximum number of characters which can be copied into the string buffer. The buffer must be large enough to store the complete list of links and the terminating null character. If this parameter is zero, the list of links are not copied to the buffer, but the function will return the number of characters required to store the complete list of links, not including the terminating null. If the ***lpszLinkList*** parameter is NULL, this parameter must be zero.

lpnLinkCount

A pointer to an integer which will contain the number of matching links extracted from the current message part. If this information is not required, this parameter can be NULL. This value will be

initialized to zero and updated with the link count if the function is successful.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer provided. If the *nMaxLength* parameter is zero, the return value specifies the number of characters required to store the list of links, not including the terminating null character. If you use this function to determine the required buffer size, allocate memory for the returned character count plus one additional character for the terminating null. If the function fails, it will return zero. To get extended error information, call **MimeGetLastError**. The last error code will be set to NO_ERROR if the parameters are valid but there are no links found in the current message part.

Remarks

The **MimeGetMessageLinks** function enables you to obtain a list of the hyperlinks which are found in the current message part. The *dwLinkTypes* parameter allows you to specify the types of links you would like to extract. The message contents are decoded and converted to UTF-8 before searching for matching links. If the message content uses HTML markup, it is possible for this function to return links in the HTML header as well as the body. Messages which are composed using Outlook will typically have links to XML namespaces like `http://www.w3.org/TR/REC-html40`. You can omit these links by not including `MIME_LINK_HTTP` in the link type bitmask.

This function can only be used to extract links from message text and cannot be used with file attachments or encoded binary data. For example, if the current message part contains encoded image data, this function will return zero. This function ignores any links that use URL schemes not listed above, such as `mailto:` or `data:`. If you have a need to find a different type of link, you can use the **MimeFindMessageText** function to search the message using a regular expression that matches your specific requirements.

When calling this function, it is generally preferable to dynamically allocate the string buffer rather than using a fixed-length buffer, which may be too small to hold all links or unnecessarily large if the message contains few or no links. This requires calling the function twice: first with *lpszLinkList* set to NULL and *nMaxLength* set to zero. The return value from this call will indicate the number of characters needed to store the list of links, excluding the terminating null character. You can then allocate a buffer of the appropriate size and call the function again to retrieve the actual list. When using this two-step approach, ensure that the *dwLinkTypes* value remains the same for both calls to avoid inconsistent results.

Example

```
hMessage = MimeOpenMessage(TEST_HTMLMSG, MIME_OPEN_DEFAULT);

if (hMessage == INVALID_MESSAGE)
{
     // Unable to open the message file
     return;
}

// Get the number of parts in a multipart message; if there is only
// one part then this is an simple RFC 822 formatted message
INT nPartCount = MimeGetMessagePartCount(hMessage);

for (INT nMessagePart = 0; nMessagePart < nPartCount; nMessagePart++)
{
    // Set the current message part
    MimeSetMessagePart(hMessage, nMessagePart);
```

```
    // If this message part is plain text or HTML, search for links
    if (MimeGetContentType(hMessage) == MIME_CONTENT_TEXT)
    {
         // Determine if there are any links in this message part and
         // how large our string buffer needs to be
         LONG cchLinkList = MimeGetMessageLinks(hMessage, MIME_LINK_HTTPS, NULL, 0,
NULL);

         // Continue to the next message part if there are no links
         if (!cchLinkList)
             continue;

         // Allocate memory for our string buffer with space for the
         // terminating null character
         LPTSTR pszLinkList = (LPTSTR)malloc((cchLinkList + 1) * sizeof(TCHAR));
         if (pszLinkList == NULL)
         {
              // Unable to allocate memory
              break;
         }

         // Return the list of links in our string buffer
         LONG nLinkCount = 0;
         cchLinkList = MimeGetMessageLinks(
                             hMessage,
                             MIME_LINK_HTTPS,
                             pszLinkList,
                             cchLinkList + 1,
                             &nLinkCount);

         if (cchLinkList > 0)
         {
            _tprintf(_T("Message part %d has %d secure links\n"), nMessagePart,
nLinkCount);

            LPTSTR pszLink = pszLinkList;
            LONG nIndex = 0;

            // Move through the list of links, replacing each space with
            // a null character to indicate the end of the link
            while (pszLink != NULL && *pszLink != '\0')
            {
                LPTSTR pszNextLink = _tcschr(pszLink, _T(' '));

                // If we found a space, what follows is the next link in the list
                if (pszNextLink != NULL)
                    *pszNextLink++ = '\0';

                _tprintf(_T("\tLink %d: %s\n"), ++nIndex, pszLink);
                pszLink = pszNextLink;
            }
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFilterMessageText, MimeFindMessageText, MimeGetContentType, MimeGetMessageText, MimeOpenMessage, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessagePart Function

```
INT WINAPI MimeGetMessagePart(
   HMESSAGE hMessage
);
```

The **MimeGetMessagePart** function returns the current message part index for the specified message.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the message part index. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCreateMessagePart, MimeDeleteMessagePart, MimeGetMessagePartCount, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessagePartCount Function

```
INT WINAPI MimeGetMessagePartCount(
    HMESSAGE hMessage
);
```

The **MimeGetMessagePartCount** function returns the total number of message parts for the specified message. Each message consists of at least one part.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the number of message parts. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCreateMessagePart, MimeGetMessagePart, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageSender Function

```
INT WINAPI MimeGetMessageSender(
    HMESSAGE hMessage,
    LPCTSTR lpszSender,
    INT nMaxLength
);
```

The **MimeGetMessageSender** function returns the email address of the message sender in the specified string buffer.

Parameters

hMessage

Handle to the message.

lpszSender

A null-terminated string buffer that will contain the email address of the message sender when the function returns.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. It is recommended that the maximum length be at least 64 characters.

Return Value

If the function succeeds, the return value is the number of bytes copied to the string buffer. If an error occurs, the function will return MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function attempts to determine the email address of the sender that originated the message. It will first check for the presence of a Sender or X-Sender header value. If these headers are not defined, it will use the value of the From header field. It will only return successfully if a valid email address can be found.

If the function succeeds, the string buffer that is provided will only contain an email address. It will not contain the display name of the user associated with the address or any extraneous comments that are included in quotes or parenthesis.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageHeaders, MimeGetFirstMessageHeader, MimeGetMessageHeaderEx, MimeGetMessagePart, MimeGetNextMessageHeader, MimeParseHeader, MimeSetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageSize Function

```
BOOL WINAPI MimeGetMessageSize(
   HMESSAGE hMessage,
   DWORD dwOptions,
   LPDWORD lpdwMessageSize
);
```

The **MimeGetMessageSize** function returns the size of the complete message in bytes.

Parameters

hMessage

Handle to the message.

dwOptions

An unsigned integer which specifies how the size of the message should be calculated, based on what header fields should be included. These are the same options used when exporting a message to a file or memory buffer. The following values may be combined using a bitwise Or operator:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.

lpdwMessageSize

A pointer to an unsigned integer value which will contain the size of the message if the function succeeds, or a value of zero if the function fails. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If an error occurs, the function will return zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function returns the size of the complete message, including all headers, the message body and any attachments. It can be used to determine the minimum amount of memory that should be allocated to export the message to a memory buffer using the **ExportMessage** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeExportMessage, MimeExportMessageEx, MimeGetMessageText, MimeImportMessage, MimeImportMessageEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageText Function

```
LONG WINAPI MimeGetMessageText(
    HMESSAGE hMessage,
    LONG nOffset,
    LPTSTR lpszMessage,
    LONG nMaxLength
);
```

The **MimeGetMessageText** function copies the text of the current message part to the specified buffer.

Parameters

hMessage

Handle to the message.

nOffset

The byte offset from the beginning of the message. A value of zero specifies the first character in the body of the message part.

lpszMessage

The string buffer that will contain a copy of the message text when the function returns. Note that the buffer will typically contain multiple lines of text. If this parameter is NULL the function will return the number of characters that would be copied.

nMaxLength

The maximum number of bytes to copy into the buffer. The size of the buffer provided must be larger than the content length for the current message part. If this parameter is zero, the text will not be copied.

Return Value

If the function succeeds, the return value is number of bytes copied into the buffer. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

If your project targets a multi-byte character set, this function will always return the message contents as UTF-8 text, regardless of the original character set specified in the message itself. This ensures that characters in the original text are preserved, regardless of the default ANSI code page on the local system. It is recommended you build your project to use Unicode whenever possible. If your application must use ANSI, you can use the **MimeLocalizeText** method to convert the Unicode text to a specific character set.

You should not determine the maximum size of the output buffer using the **MimeGetContentLength** function. That function returns the content size in bytes as it is stored in the message, and does not account for any character encoding or Unicode conversion which may be required. The content length can be used to estimate the amount of text stored in the message part, but you should always allocate a buffer which is larger than the length specified in the message.

If the *nMaxLength* parameter does not specify a buffer size large enough to store the contents of the current message part, this function will fail and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. Your application must ensure the buffer is large enough to contain the complete message text and a terminating null character. If you wish to determine the actual length of the decoded message text, you can specify the *lpszMessage* parameter as NULL and *nMaxLength*

as zero. This will cause the function to return the number of characters in the current message part, not including the terminating null. When allocating a buffer on the heap, make sure you include additional space for the terminating null character. See the example below.

If the current message part contains HTML markup and you want to display it as plain text to the user, you can use the **MimeFilterMessageText** function. It can remove markup from the message text, as well as remove potentially dangerous hyperlinks.

Example

```
if (MimeGetContentType(hMessage) == MIME_CONTENT_TEXT)
{
    // Get the size of the decoded/converted message text
    LONG cchMessage = MimeGetMessageText(hMessage, 0, NULL, 0);
    if (cchMessage < 1)
    {
        _tprintf(_T("Message part %d does not contain any text\n"), nMessagePart);
        return;
    }

    // Allocate a buffer for the message text +1 for the null character
    LPTSTR pszMessage = (LPTSTR)malloc((cchMessage + 1) * sizeof(TCHAR));
    if (pszMessage == NULL)
        return;

    // Copy the message contents to the buffer
    cchMessage = MimeGetMessageText(hMessage, 0, pszMessage, cchMessage + 1);
    if (cchMessage > 0)
    {
        // Write the contents of the message part to the console
        fwrite(pszMessage, sizeof(TCHAR), cchMessage, stdout);
        fflush(stdout);
    }

    // Free the memory allocated for our text buffer
    free(pszMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAppendMessageText, MimeClearMessageText, MimeCompareMessageText, MimeExportMessage, MimeFilterMessageText, MimeGetContentLength, MimeImportMessage, MimeLocalizeText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetMessageVersion Function

```
LPCTSTR WINAPI MimeGetMessageVersion(
    HMESSAGE hMessage
);
```

The **MimeGetMessageVersion** function returns a pointer to a string which contains the MIME version header value for the specified message.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is a pointer to the MIME version. If the version is not specified in the current message, the function will return NULL. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetMessageHeader, MimeSetMessageVersion

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetNextMessageHeader Function

```
BOOL WINAPI MimeGetNextMessageHeader(
    HMESSAGE hMessage,
    LPTSTR lpszHeader,
    INT cchMaxHeader,
    LPTSTR lpszValue,
    INT cchMaxValue
);
```

The **MimeGetNextMessageHeader** function returns the header field name and value for the next header in the current message part. This function is typically used in conjunction with **MimeGetFirstMessageHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

hMessage

Handle to the message.

lpszHeader

The string buffer that will contain the name of the next header in the current message part when the function returns. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

The string buffer that will contain the value of the next header in the current message part when the function returns. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the ***lpszValue*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is non-zero. If no more headers exist for the current message part, or the handle to the message is invalid, the function will return zero. To get extended error information, call **MimeGetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **MimeSetMessagePart** function.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined, with the following exception:

If an imported message has multiple Received headers, then those headers will be returned by **MimeGetNextMessageHeader** in the order in which they appeared in the original message. Note

that if **MimeGetMessageHeader** is used to retrieve the Received header, the first Received header in the message will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeEnumMessageHeaders, MimeGetFirstMessageHeader, MimeGetMessagePart, MimeParseHeader, MimeSetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetStoredMessage Function

```
HMESSAGE WINAPI MimeGetStoredMessage(
    HMESSAGESTORE hStorage,
    LONG nMessageId,
    DWORD dwOptions
);
```

The **MimeGetStoredMessage** function retrieves a message from the specified message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which specifies the message number that should be retrieved. The first message in the message store has a value of one.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
A shared message handle returned by the function. The contents of this message will be overwritten each time this function is called.	
MIME_COPY_STORED_MESSAGE	A message handle is allocated for a copy of the message that is retrieved from the message store.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is INVALID_MESSAGE. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetStoredMessage** function returns a message handle for the specified message in the message store. If no options are specified, a temporary message handle is returned that is only valid until the next message is retrieved. The application can use message handle with any function except for the **MimeCloseMessage** function because it does not own the message handle. If a multithreaded application changes the contents of the temporary message, it will change for all other threads that have obtained a message handle using this function.

If the application must have a unique copy of the message, the MIME_COPY_STORED_MESSAGE option should be specified. Instead of returning a handle to a shared message, the message is duplicated and a handle to that copy of the message is returned. If this option is used, the application must call **MimeCloseMessage** to release the memory allocated for the message.

Example

```
HMESSAGE hMessage = INVALID_MESSAGE;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
```

```
LONG nMessageId = 1;

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = MimeFindStoredMessage(hStorage,
                                       nMessageId,
                                       lpszHeader,
                                       lpszAddress,
                                       MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        // Get a handle to the message that was found
        hMessage = MimeGetStoredMessage(hStorage, nMessageId, 0);

        if (hMessage != INVALID_MESSAGE)
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = MimeExportMessage(hMessage, szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeFindStoredMessage, MimeGetStoredMessageCount, MimeDeleteStoredMessage, MimeStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeGetStoredMessageCount Function

```
LONG WINAPI MimeGetStoredMessageCount(
    HMESSAGESTORE hStorage,
    LPLONG lpnLastMessage
);
```

The **MimeGetStoredMessageCount** function returns the number of messages in the specified message store.

Parameters

hStorage

Handle to the message store.

lpnLastMessage

A pointer to an integer which will contain the message number for the last message in the storage file. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is the number of messages in the message store. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetStoredMessageCount** function returns the number of messages in the message store. It is important to note that does not count those messages which have been marked for deletion. This means that the value returned by this function will decrease as messages are deleted.

The message number returned in the ***lpnLastMessage*** parameter will specify the total number of messages in the message store, including deleted messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeFindStoredMessage, MimeGetStoredMessage, MimeDeleteStoredMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeImportMessage Function

```
BOOL WINAPI MimeImportMessage(
   HMESSAGE hMessage,
   LPCTSTR lpszFileName
);
```

The **MimeImportMessage** function reads the specified file and replaces the current message with its contents.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the text file to import.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function will delete the contents of the current message before importing the new message from the specified file. Unlike the **MimeOpenMessage** function, this function is intended to be used when you wish to replace an existing message with the contents of another MIME structured file.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeExportMessage, MimeExportMessageEx, MimeExtractFile, MimeImportMessageEx, MimeInitialize, MimeOpenMessage, MimeParseText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeImportMessageEx Function

```
BOOL WINAPI MimeImportMessageEx(
   HMESSAGE hMessage,
   DWORD dwImportMode,
   DWORD dwImportOptions,
   LPVOID lpvMessage,
   DWORD dwMessageSize
);
```

The **MimeImportMessageEx** function imports the contents of a message from a file, string or memory handle.

Parameters

hMessage

Handle to the message.

dwImportMode

An unsigned integer which specifies how the message contents will be imported. It may be one of the following values:

Value	Description
MIME_IMPORT_DEFAULT	The default import mode. If the ***lpvMessage*** parameter is NULL, then the contents of the message will be imported from the system clipboard. Otherwise, the ***lpvMessage*** parameter is a pointer to a string which specifies the name of a file that contains the message. The ***dwMessageSize*** parameter is ignored.
MIME_IMPORT_FILE	The ***lpvMessage*** parameter is a pointer to a string which specifies the name of the file that contains the message. If the file does not exist, or is not a regular text file, an error will occur. The ***dwMessageSize*** parameter is ignored.
MIME_IMPORT_CLIPBOARD	The contents of the message is imported from the system clipboard. The ***lpvMessage*** parameter is ignored. The ***dwMessageSize*** parameter is ignored.
MIME_IMPORT_MEMORY	The contents of the message is imported from a local buffer. The ***lpvMessage*** parameter must point to a byte array which contains the message to be imported. The ***dwMessageSize*** parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.
MIME_IMPORT_HGLOBAL	The contents of the message is imported from a global memory buffer. The ***lpvMessage*** parameter must be a global memory handle which contains the message. The ***dwMessageSize*** parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.

dwImportOptions

An unsigned integer which specifies how the message will be imported:

Value	Description
MIME_OPTION_DEFAULT	The default import options. Currently this is the only valid value for this parameter and applications should always specify this constant.

lpvMessage

A null-terminated string, a byte buffer or a global memory handle. The ***dwImportMode*** parameter determines how this pointer is used by the function.

dwMessageSize

An unsigned integer value which specifies the size of the message to import. This parameter is only used when importing a message from a memory buffer. The message size is determined automatically when the message is imported from a file or the system clipboard.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Example

```
// Imports a message using an HGLOBAL handle which references
// a block of memory containing an email message
bResult = MimeImportMessageEx(hMessage,
                              MIME_IMPORT_HGLOBAL,
                              MIME_OPTION_DEFAULT,
                              (LPVOID)hgblMessage,
                              0);

if (bResult)
{
    // The message has been successfully imported
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCreateMessage, MimeExportMessage, MimeExportMessageEx, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeInitialize Function

```
BOOL WINAPI MimeInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **MimeInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **MimeInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **MimeUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **MimeInitialize** or **MimeUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCloseMessage, MimeComposeMessage, MimeCreateMessage, MimeUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeIsContentType Function

```
BOOL WINAPI MimeIsContentType(
   HMESSAGE hMessage,
   INT nMessagePart,
   LPCTSTR lpszContentType
);
```

The **MimeIsContentType** function checks if the content type for the message matches the specified content type.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value which specifies which part of the message to compare against the content type. A value of zero returns compares the content type against the main message header, while a value greater than zero compares the content type against the content type for a specific part of a multipart message. A value of MIME_CURRENT_PART (-1) specifies that the content type should be compared against the current message part.

lpszContentType

Pointer to a string which specifies the content type to compare against. This parameter cannot be NULL and it must specify a valid MIME media type. It is permitted to omit the media subtype, in which case the content type match will only be made against the base type. For example, both `"application/pdf"` and `"application"` are permitted as valid content types. Note that this is a prefix match, not a wildcard pattern match.

Return Value

The function will return a non-zero value if the specified content type matches the content type for the message part. If the content types do not match or one of the parameters is invalid, the function will return zero. To get extended error information, call **MimeGetLastError**. If the parameters are valid but the content types do not match, the last error code will be set to NO_ERROR.

Remarks

The **MimeIsContentType** function is used to check the content type for a specific part of a MIME-encoded message against a known media type. Multipurpose Internet Mail Extensions (MIME) defines a standard way of classifying the content of an email message or its individual parts, such as plain text, HTML, images, or file attachments.

A MIME media type, also referred to as a content type, is typically expressed in the form `type/subtype`. For example, the content type `text/plain` identifies unformatted plain text, while `text/html` identifies HTML formatted text. The content type for each message part is typically specified in its `Content-Type` header field. This function performs a case-insensitive comparison between the message part's declared content type and the type provided by the caller. It does not consider optional parameters, such as character encoding or boundary markers. For example, a content type of `"text/html; charset=UTF-8"` will match `"text/html"`.

The following table lists some of the most common MIME media types used in email messages:

Media Type	Description

`text/plain`	Plain text with no formatting
`text/html`	HTML-formatted text, commonly used for rich email content
`multipart/alternative`	Message that contains multiple representations of the same content, such as both plain text and HTML
`multipart/mixed`	Message that includes a combination of text and attachments
`multipart/related`	Message that includes related resources, such as inline images referenced by HTML content
`image/jpeg`	JPEG image
`image/png`	PNG image
`application/pdf`	PDF document
`application/octet-stream`	Generic binary data, typically used for file attachments

To obtain the full content type string for a message part, including any optional parameters, use the **MimeGetMessageHeaderEx** function with the header name `"Content-Type"`.

Example

```
// Search the message for a PDF attachment by checking the content type
// for each part of a multipart message.
//
// If the content type matches "application/pdf" then that message part
// should contain a PDF attachment. We prompt the user where to store
// the file and then exit the loop.

INT nParts = MimeGetMessagePartCount(hMessage);

for (INT nMessagePart = 1; nMessagePart < nParts; nMessagePart++)
{
    if (MimeSetMessagePart(hMessage, nMessagePart) == MIME_ERROR)
        break;

    if (MimeIsContentType(hMessage, MIME_CURRENT_PART, _T("application/pdf")))
    {
        TCHAR szAttachment[MAX_PATH] = _T("");

        // Get the name of the file attached to the message. This can be
        // useful for display purposes, but exercise caution if you want
        // to use that name to store the file locally.
        if (MimeGetAttachedFileName(hMessage, szAttachment, MAX_PATH))
        {
            _tprintf(_T("Found attachment \"%s\" in part %d\n"), szAttachment,
nMessagePart);

            // The recommended approach is to prompt your user for where
            // they would like you to store store the file attachment;
            // avoid using the remote filename directly without validating
            // or prompting the user.
            TCHAR szLocalFile[MAX_PATH] = _T("");

            OPENFILENAME ofn = {0};
            ofn.lStructSize = sizeof(OPENFILENAME);
            ofn.hwndOwner = NULL; // No owner window
```

```
            ofn.lpstrFilter = "Text Files (*.txt)\0*.txt\0All Files (*.*)\0*.*\0";
            ofn.lpstrFile = szLocalFile;
            ofn.nMaxFile = MAX_PATH;
            ofn.lpstrTitle = "Select File to Save";
            ofn.Flags = OFN_OVERWRITEPROMPT;

            // If the user cancels the dialog, we are finished
            if (!GetSaveFileName(&ofn))
                 break;

            // Extract the attachment to the file name specified by the user
            if (!MimeExtractFile(hMessage, szLocalFile))
            {
                DWORD dwError = MimeGetLastError();
                _tprintf(_T("Failed to extract file (Error 0x%08lX)\n"), dwError);
            }
        }
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCompareMessageText, MimeExtractFile, MimeFindAttachment, MimeGetAttachedFileName, MimeGetMessageHeaderEx, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeLocalizeText Function

```
LONG WINAPI MimeLocalizeText(
   UINT nCharacterSet,
   LPCTSTR lpszInput,
   LONG cchInput,
   LPBYTE lpOutput,
   LONG cbOutput
);
```

The **MimeLocalizeText** function converts a Unicode string to its ANSI equivalent using the specified character set.

Parameters

nCharacterSet

A numeric identifier which specifies the character set to use when localizing the text. A value of zero specifies the locale for the current thread should be used when localizing the text.

lpszInput

A null-terminated string which contains the Unicode text which should be localized. If the ANSI version of this function is called, the input text must be in UTF-8 format or the function will fail. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be localized. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpOutput

A buffer which will contain the localized ANSI version of the input text. This parameter cannot be a NULL pointer.

cbOutput

An integer value which specifies the maximum number of bytes which can be copied into the output buffer. The buffer must be large enough to store all of the localized text. This value must be greater than zero.

Return Value

If the input text can be successfully localized, the return value is the number of bytes copied into the output buffer. If the function returns zero, then no text was localized. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeLocalizeText** function enables the application to localize a Unicode string, returning the ANSI version of that text using the specified character set. Because library handles all text as Unicode internally, the ANSI functions in this library will always return UTF-8 encoded text. This function allows you to easily convert that UTF-8 text to another supported character set.

If the ANSI version of this function is called, the input text must use UTF-8 character encoding or the function will fail with the last error set to ST_ERROR_INVALID_CHARSET.

If the *nCharacterSet* parameter is MIME_CHARSET_DEFAULT or MIME_CHARSET_UNKNOWN the input text will be converted to the default ANSI code page for the current thread locale. If there are characters in the Unicode input text which cannot be converted to an ANSI equivalent using the

specified character set, those characters will be replaced by the default character for your locale, typically a question mark symbol. You cannot specify MIME_CHARSET_UTF16 as the character set.

This function will always attempt to ensure that the output buffer is terminated with an extra null byte so it is easier to work with as a standard C-style null-terminated string. However, if the output buffer is not large enough to accommodate the extra null byte, it will not be copied. It is always recommended that your output buffer be somewhat larger than the length of the original input text to account for multi-byte character sequences. If the output buffer is not large enough to contain the entire localized text, no bytes will be copied to the output buffer and the function will fail with the last error set to ST_ERROR_BUFFER_TOO_SMALL.

This function is only required if your application needs to localize the UTF-8 text returned by another function and convert it to a specific 8-bit ANSI character set. For example, if you have an application which calls the ANSI version of **MimeGetMessageText**, it will return the message contents as UTF-8 text. If you need to display that text as localized ANSI, you can call this function to convert the UTF-8 text to your current locale or a specific character set.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeDecodeTextEx, MimeEncodeText, MimeEncodeTextEx, MimeGetMessageText, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeNormalizeText Function

```
LONG WINAPI MimeNormalizeText(
    LPCTSTR lpszInput,
    LONG cchInput,
    LPTSTR lpszOutput,
    LONG nMaxLength
);
```

The **MimeNormalizeText** function normalizes the input text to replace confusable Unicode homoglyphs, remove symbols and trim whitespace.

Parameters

lpszInput

A null-terminated string which contains the Unicode text that will be normalized. The input may only contain text and cannot include non-printable control characters. If this parameter is NULL or if the input contains binary data, this function will fail. If the ANSI version of this function is called, the input string must be contain ASCII or UTF-8 encoded Unicode characters.

cchInput

An integer which specifies the number of characters in the input string. If this parameter is -1, the length of the string will be determined by counting the number of characters up to a terminating null character.

lpszOutput

A null-terminated string buffer which will contain the normalized text. The buffer must be large enough to contain the entire string and a terminating null character or the function will fail. If this parameter is NULL and the ***nMaxLength*** parameter is zero, the function will return the number of characters required to store the normalized text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the ***lpszOutput*** string buffer. If this value is zero, no characters will be copied to the output buffer and the function will return the length of the filtered output, not including the terminating null character. This can be used to determine how much memory should be allocated to store the filtered text.

Return Value

If the function succeeds, it will return the number of characters copied into the output buffer, not including the terminating null character. If the ***lpszOutput*** parameter is NULL and ***nMaxLength*** is zero, the function will return the number of characters in the filtered text, not including the terminating null. If the function fails, or there is no text in the input buffer, the function will return zero.

Remarks

The **MimeNormalizeText** function performs a multi-stage normalization and cleanup of Unicode text, intended to simplify and standardize input that may contain visually similar (confusable) characters, diacritical marks, or inconsistent whitespace. It applies NFKD (Compatibility Decomposition) normalization converting characters into their canonical forms and then replaces any confusable Unicode characters with their ASCII equivalents based on an internal mapping table. This function is similar to calling the **NormalizeString** function in the Windows API, however it performs additional checks and is more strict with how the input is validated.

If the **MimeNormalizeTextA** (ANSI) version of this function is called, the input string is checked to

ensure that it only contains valid UTF-8 encoding and will reject strings which contain invalid sequences. For example, it will reject multibyte UTF-8 sequences which are over-long or truncated, invalid continuation sequences or characters outside of valid Unicode code points. You should never provide a localized ANSI string composed using the current code page. If you wish to normalize a localized string, you should convert it to Unicode using the **MultiByteToWideChar** function and then call **MimeNormalizeTextW**.

If the **MimeNormalizeTextW** (Unicode) version of this function is called, the input string is validated to ensure that it only contains valid UTF-16 encoding and will reject strings which contain invalid sequences. This includes checking to ensure there are no unpaired surrogates, invalid code points, unprintable control characters or private-use characters in the string. If the input string does not contain valid Unicode character sequences, the function will return zero and the last error code will be set to ST_ERROR_INVALID_UNICODE.

This function helps mitigate issues with characters that are visually indistinguishable from standard Latin letters, such as using the Cyrillic "a" (U+0430) instead of the Latin "a" (U+0061). It also removes symbols (such as emojis, flags and pictographs) which can confuse pattern matching and removes combining marks that can be used to visually modify characters or obscure text. Unicode whitespace is trimmed from both ends of the string, including zero-width and non-breaking spaces.

This function is particularly useful in security-sensitive applications where input normalization is critical, such as sanitizing user-provided domain names, filenames, or login credentials to prevent spoofing, phishing, or bypassing filters through the use of homoglyphs or invisible characters.

If you want a normalized version of the text in a message, you can use the **MimeFilterMessageText** function. It performs many of the same normalizations that this function does, but allows for more flexibility through options which allow you to control how the message text is processed and filtered. It is particularly useful for filtering extraneous HTML from the message, enabling you to display a version of the message that is safer for the user (such as removing unsafe links from the message contents).

This function is deliberately biased toward security rather than readability and can affect the output when applied to text containing legitimate non-Latin characters. It is intended for use on text that is expected to contain only printable ASCII characters. For example, it will filter and normalize text containing certain Greek or Cyrillic characters that could be used to spoof ASCII letters or URL delimiters. While this helps mitigate obfuscated or malicious content, it can also make legitimate text in other languages unreadable.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetContentType, MimeIsContentType, MimeCompareMessageText, MimeFilterMessageText, MimeFindMessageText, MimeGetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeOpenMessage Function

```
HMESSAGE WINAPI MimeOpenMessage(
   LPCTSTR lpszFileName,
   DWORD dwOptions
);
```

The **MimeOpenMessage** function opens and reads the specified file and returns a handle to the message.

Parameters

lpszFileName

Pointer to a string which specifies the name of the text file to open. This parameter cannot be NULL.

dwOptions

An integer which specifies options for opening the message file. This parameter is reserved for future use and should always be zero in the current implementation.

Value	Description
MIME_OPEN_DEFAULT	The default mode for opening the message file. Currently, this is the only option which should be specified.

Return Value

If the function succeeds, the return value is a handle to the message which can be used with other functions. If the function fails, the return value is INVALID_MESSAGE. To get extended error information, call **MimeGetLastError**.

Remarks

This function opens the specified file, reads the contents into memory and then parses the message. The file must be a text file which uses the standard MIME message structure. It may contain multiple parts, including file attachments and alternative representations of the same message. This function cannot be used to open messages stored in a binary format.

If the file path includes environment variables enclosed in percentage (%) symbols, those values will be expanded based on the environment variables defined for the current user using the **ExpandEnvironmentStrings** function in the Windows API. Note that environment variable names are not case-sensitive.

The maximum size of an individual message is 2 GiB (approximately 2 billion characters). This limitation is implemented as a safety check to prevent opening very large files which are unlikely to contain a valid MIME formatted message. Most services impose much lower limits on the maximum size of an individual mail message.

Note: This function can only be used to open and parse messages structured according to the MIME (Multipurpose Internet Mail Extensions) standard, such as plain text `.eml` or `.txt` files. It cannot open messages stored in proprietary or binary formats used by certain email clients. For example, it does not support Microsoft Outlook message files such as `.msg` or mailbox files like `.pst` and `.ost`, nor does it support formats used by other clients such as Thunderbird's `.msf` (mail summary files) or the mbox family (`.mbx`, `.mbox`) when stored as binary containers. Attempting to open these file types will result in an error. If you need to work with messages exported from Outlook or other proprietary clients, first save or export them in the standard text format before using this function.

If you wish to create a new message rather than opening an existing message, use the **MimeComposeMessage** function. When you are finished accessing the contents of the message, use the **MimeCloseMessage** function to close the message handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachFile, MimeCloseMessage, MimeComposeMessage, MimeExportMessage, MimeExtractFile, MimeImportMessage, MimeInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeOpenMessageStore Function

```
HMESSAGESTORE WINAPI MimeOpenMessageStore(
    LPCTSTR lpszFileName,
    DWORD dwOpenMode
);
```

The **MimeOpenMessageStore** function opens the specified message storage file.

Parameters

lpszFileName

A null-terminated string which specifies the name of the storage file.

dwOpenMode

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
MIME_STORAGE_READ	The message store will be opened for read access. The contents of the message store can be accessed, but cannot be modified by the process unless it has also been opened for writing.
MIME_STORAGE_WRITE	The message store will be opened for writing. This mode also implies read access and must be specified if the application needs to modify the contents of the message store.
MIME_STORAGE_CREATE	The message store will be created if the storage file does not exist. If the file exists, it will be truncated. This mode implies read and write access.
MIME_STORAGE_LOCK	The message store will be opened so that it may only be accessed and modified by the current process.
MIME_STORAGE_COMPRESS	The contents of the message store are compressed. This option is automatically enabled if a compressed message store is opened for reading or writing.
MIME_STORAGE_MAILBOX	The message store should use the UNIX mbox format when reading and storing messages. This option is provided for backwards compatibility and is not recommended for general use.

Return Value

If the function succeeds, the return value is a handle to the message store. If the function fails, the return value is INVALID_MESSAGESTORE. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeOpenMessageStore** function opens a message storage file which contains one or more messages. If the storage file is opened for read access, the application can search the message store and extract messages but it cannot add or delete messages. To add new messages or delete existing messages from the store, it must be opened with write access.

The message store is designed to be a simple, effective way to store multiple messages together in a single file. When the message store is opened, the contents are indexed in memory. Although there is no specific limit to the number of messages that can be stored, there must be sufficient memory available to build an index of each message and its headers. If the application must store and manage a very large number of messages, it is recommended that you use a database rather than a flat-file message store.

Message Store Format

Each message is prefixed by a control sequence of five ASCII 01 characters followed by an ASCII 10 and ASCII 13 character. The messages themselves are stored unmodified in their original text format. The length of each message is calculated based on the location of the control sequence that delimits each message, and explicit message lengths are not stored in the file. This means that it is safe to manually change the message contents, as long as the message delimiters are preserved.

If the message store is compressed, the contents of the storage file are expanded when the file is opened and then re-compressed when the storage file is closed. Using the MIME_STORAGE_COMPRESS option reduces the size of the storage file and prevents the contents of the message store from being read using a text file editor. However, enabling compression will increase the amount of memory allocated by the library and can increase the amount of time that it takes to open and close the storage file.

The library also has a backwards compatibility mode where it will recognize storage files that use the UNIX mbox format. While this format is supported for accessing existing files, it is not recommended that you use it when creating new message stores or adding messages to an existing store. There are a number of different variants on the mbox format that have been used by different Mail Transfer Agents (MTAs) on the UNIX platform. For example, the mboxrd variant looks identical to the mboxcl2 variant, and they are programmatically indistinguishable from one another, but they are not compatible. For this reason, the use of the mbox format is strongly discouraged.

Example

```
HMESSAGE hMessage;

// Compose a new message
hMessage = MimeComposeMessage(lpszSender,
                              lpszRecipient,
                              NULL,
                              lpszSubject,
                              lpszMessage,
                              NULL,
                              MIME_CHARSET_DEFAULT,
                              MIME_ENCODING_DEFAULT);

if (hMessage != INVALID_MESSAGE)
{
    HMESSAGESTORE hStorage;

    // Open the message storage file
    hStorage = MimeOpenMessageStore(lpszFileName, MIME_STORAGE_WRITE);

    if (hStorage == INVALID_MESSAGESTORE)
    {
        // Close the message and return if we are unable to
        // open the storage file
        MimeCloseMessage(hMessage);
        return;
```

```
    }

    // Store a copy of the message in the message store
    nMessageId = MimeStoreMessage(hStorage, hMessage, 0);

    if (nMessageId == MIME_ERROR)
    {
        // We were unable to store the message
    }

    // Close the message store
    MimeCloseMessageStore(hStorage);

    // Close the message that was created
    MimeCloseMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCloseMessageStore, MimeFindStoredMessage, MimeGetStoredMessage, MimeGetStoredMessageCount, MimePurgeMessageStore

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeParseAddress Function

```
INT WINAPI MimeParseAddress(
   LPCTSTR lpszString,
   LPCTSTR lpszDomain,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **MimeParseAddress** function parses a string for an email address, copying the address to the specified buffer.

Parameters

lpszString

A null-terminated string which contains the email address to parse.

lpszDomain

A null-terminated string which specifies a default domain for the address. This parameter may be NULL or point to an empty string if no default domain is required.

lpszAddress

A null-terminated string buffer which will contain the parsed email address when the function returns. It is recommended that this buffer be at least 128 characters in length.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the address. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeParseAddress** function is useful for parsing the email addresses that may be specified in various header fields in the message. In many cases, the addresses have additional comment characters which are not part of the address itself. For example, one common format used is as follows:

"User Name" <user@domain.com>

In this case, the email address is enclosed in angle brackets and the name outside of the brackets is considered to be a comment which is not part of the address itself. Another common format is:

user@domain.com (User Name)

In this case, there is the address followed by a comment which is enclosed in parenthesis. The **MimeParseAddress** function recognizes both formats, and when passed either string, would return the following address:

user@domain.com

If there was no domain specified in the address, that is just a user name was specified, then the value the *lpszDomain* parameter is added to the address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeExportMessage, MimeExtractFile, MimeImportMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeParseDate Function

```
BOOL WINAPI MimeParseDate(
   LPCTSTR lpszDate,
   BOOL bLocalize,
   LONG *lpnSeconds,
   LONG *lpnTimezone
);
```

The **MimeParseDate** function parses a message date string, returning the number of seconds since 1 January 1970 and the difference in seconds between the specified timezone and Coordinated Universal Time.

Parameters

lpszDate

A null-terminated string which specifies the date to be parsed. The date string must be in the standard format defined by RFC 822.

bLocalize

A boolean flag which determines if the date should be localized to the current timezone, regardless of the timezone specified in the date string. A non-zero value specifies the timezone for the local system will be used, adjusted for daylight savings time if applicable.

lpnSeconds

A pointer to a long integer which is set to the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C/C++ time functions. This pointer may be NULL, in which case the parameter is ignored.

lpnTimezone

A pointer to a long integer which is set to the difference in seconds between the specified date's timezone and coordinated universal time (also known as Greenwich Mean Time). This pointer may be NULL, in which case the parameter is ignored.

Return Value

If the date could be successfully parsed, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This is not a general purpose date parsing function, and it may not be capable of parsing dates for a specific locale. The date and time should be in a standard format as outlined in RFC 822, which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **MimeFormatDate** function.

If the date and time does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you use this function to parse input from a user, because in most cases they will not provide a timezone and will assume the date and time they enter is for their current timezone.

The value of the ***bLocalize*** parameter will only change the number of seconds offset by the current timezone and does not affect the value returned in the ***lpnSeconds*** parameter. If the date and time is localized, the timezone offset will be adjusted for daylight savings if it was in effect at the time.

Developers are strongly encouraged to use the **MimeParseDate64** function for all new development.

This function represents date and time values as 32-bit integers, which limits the representable range to dates before 19 January 2038 (the so-called "Year 2038 problem"). After this date, 32-bit time values will overflow, resulting in incorrect date calculations and potential failures in software that processes email message dates. By contrast, the 64-bit versions use 64-bit integer values, matching the standard **time_t** type used by modern C/C++ libraries. This not only eliminates the Year 2038 limitation but also ensures compatibility with current and future systems, allowing correct handling of dates both far in the past and well into the future. Whenever possible, applications should transition to these 64-bit functions to ensure long-term reliability and portability.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFormatDate, MimeFormatDate64, MimeGetMessageDate, MimeParseDate64, MimeSetMessageDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeParseDate64 Function

```
BOOL WINAPI MimeParseDate64(
   LPCTSTR lpszDate,
   BOOL bLocalize,
   LONGLONG *lpnSeconds,
   LONG *lpnTimezone
);
```

The **MimeParseDate64** function parses a message date string, returning the number of seconds since 1 January 1970 and the difference in seconds between the specified timezone and Coordinated Universal Time.

Parameters

lpszDate

A null-terminated string which specifies the date to be parsed. The date string must be in the standard format for mail messages.

bLocalize

A boolean flag which determines if the date should be localized to the current timezone, regardless of the timezone specified in the date string. A non-zero value specifies the timezone for the local system will be used, adjusted for daylight savings time if applicable.

lpnSeconds

A pointer to a 64-bit integer which is set to the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C/C++ time functions. This pointer may be NULL, in which case the parameter is ignored.

lpnTimezone

A pointer to a 32-bit integer which is set to the difference in seconds between the specified date's timezone and coordinated universal time (also known as Greenwich Mean Time). This pointer may be NULL, in which case the parameter is ignored.

Return Value

If the date could be successfully parsed, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This is not a general purpose date parsing function, and it may not be capable of parsing dates for a specific locale. The date and time should be in a standard format as outlined in RFC 822, which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **MimeFormatDate64** function.

If the date and time does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you use this function to parse input from a user, because in most cases they will not provide a timezone and will assume the date and time they enter is for their current timezone.

The value of the *bLocalize* parameter will only change the number of seconds offset by the current timezone and does not affect the value returned in the *lpnSeconds* parameter. If the date and time is localized, the timezone offset will be adjusted for daylight savings if it was in effect at the time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFormatDate, MimeGetMessageDate, MimeSetMessageDate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeParseHeader Function

```
BOOL WINAPI MimeParseHeader(
   HMESSAGE hMessage,
   LPCTSTR lpszHeader
);
```

The **MimeParseHeader** function parses a line of text and adds the header and value to the current message.

Parameters

hMessage

Handle to the message.

lpszHeader

Pointer to a null-terminated string which contains the header and value to be added to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function is used to parse a line of text that is part of a message header. The string must consist of a header name, followed by a colon, followed by the header value. The header name may only consist of printable characters, and may not contain whitespace (space, tab, carriage return or linefeed characters).

In most cases, it is recommended you use the **MimeSetMessageHeader** function to set the value of a header.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetMessageHeader, MimeImportMessage, MimeParseText, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeParseText Function

```
BOOL WINAPI MimeParseText(
   HMESSAGE hMessage,
   LPCTSTR lpszText,
   INT cchText
);
```

The **MimeParseText** function parses the string buffer and adds the contents to the message.

Parameters

hMessage

Handle to the message.

lpszText

Pointer to a string buffer which contains the text to be added to the message contents.

cchText

The length of the specified buffer. If this value is -1 all of the characters in the string will be parsed up to the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function is useful when the application needs to parse an arbitrary block of text and add it to the specified message. If the string contains header fields, the values will be added to the message header. Once the end of the header block is detected, all subsequent text is added to the body of the message.

To open and parse a text file which contains a MIME structured email message, use the **MimeOpenMessage** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeCloseMessage, MimeImportMessage, MimeImportMessageEx, MimeOpenMessage, MimeParseHeader, MimeResetMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimePurgeMessageStore Function

```
BOOL WINAPI MimePurgeMessageStore(
   HMESSAGESTORE hStorage
);
```

The **MimePurgeMessageStore** function purges all deleted messages from the specified message store.

Parameters

hStorage

Handle to the message store.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimePurgeMessageStore** function purges all deleted messages from the message store. If the storage file has been opened in read-only mode or there are no messages marked for deletion, this function will take no action.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCopyMessageStore, MimeFindStoredMessage, MimeGetStoredMessage, MimeDeleteStoredMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeReplaceStoredMessage Function

```
BOOL WINAPI MimeReplaceStoredMessage(
    HMESSAGESTORE hStorage,
    LONG nMessageId,
    HMESSAGE hMessage,
    DWORD dwReserved
);
```

The **MimeReplaceStoredMessage** function replaces the specified message in a message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which specifies the message number that should be replaced.

hMessage

Handle to the message that will be stored.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeReplaceStoredMessage** function replaces the specified message with a new message. The message number may be a message that has been previously marked for deletion. It is important to note that the change will not be reflected in the physical storage file until it has been closed. If the application needs to replace messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Example

```
HMESSAGE hMessage;

// Compose a new message
hMessage = MimeComposeMessage(lpszSender,
                              lpszRecipient,
                              NULL,
                              lpszSubject,
                              lpszMessage,
                              NULL,
                              MIME_CHARSET_DEFAULT,
                              MIME_ENCODING_DEFAULT);

if (hMessage != INVALID_MESSAGE)
{
    HMESSAGESTORE hStorage;
    LONG nLastMessage;
    BOOL bResult;
```

```
    // Open the message storage file
    hStorage = MimeOpenMessageStore(lpszFileName, MIME_STORAGE_WRITE);

    if (hStorage == INVALID_MESSAGESTORE)
    {
        // Close the message and return if we are unable to
        // open the storage file
        MimeCloseMessage(hMessage);
        return;
    }

    if (MimeGetStoredMessageCount(hStorage, &nLastMessage) < 1)
    {
        // No messages are stored in the file
        MimeCloseMessageStore(hStorage);
        MimeCloseMessage(hMessage);
        return;
    }

    // Replace the last message in the message store
    bResult = MimeReplaceStoredMessage(hStorage, nLastMessage, hMessage, 0);

    if (bResult == FALSE)
    {
        // We were unable to replace the message
    }

    // Close the message store
    MimeCloseMessageStore(hStorage);

    // Close the message that was created
    MimeCloseMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeFindStoredMessage, MimeGetStoredMessage, MimeDeleteStoredMessage, MimeStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeResetMessage Function

```
BOOL WINAPI MimeResetMessage(
   HMESSAGE hMessage
);
```

The **MimeResetMessage** function clears the header and body of the specified message, and deletes all message parts.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeCloseMessage, MimeCreateMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetExportOptions Function

```
BOOL WINAPI MimeSetExportOptions(
    HMESSAGE hMessage,
    DWORD dwOptions
);
```

The **MimeSetExportOptions** function specifies a bitmask that describes current message export options.

Parameters

hMessage

Handle to the message.

dwOptions

Mask of attribute options. The mask is a combination of the following values:

Value	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

By default, the Received and Return-Path headers are not exported. In addition, the order of the headers in an exported message is undefined. This is reasonable behavior for most mail clients, but may not be appropriate for applications which need access to all of the header fields.

Example

```
MimeSetExportOptions(hMessage, MIME_EXPORT_OPTIONS_ALL |
                               MIME_EXPORT_OPTIONS_KEEP_ORDER);

if (MimeImportMessage(hMessage, lpszFileName))
{
    // Process the message and make any modifications
    // then write the message back out, preserving all
    // of the headers in their original order
    bResult = MimeExportMessage(hMessage, lpszFileName);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeExportMessage, MimeExportMessageEx, MimeImportMessage, MimeImportMessageEx, MimeGetExportOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetFileContentType Function

```
BOOL WINAPI MimeSetFileContentType(
    HMESSAGE hMessage,
    UINT nContentType,
    LPCTSTR lpszSubtype,
    LPCTSTR lpszExtension
);
```

The **MimeSetFileContentType** function associates a per-message content type with a given file name extension. This association is specific to the message, and is not shared by an other messages that may be opened by the process.

Parameters

hMessage

Handle to the message.

nContentType

An identifier which specifies the primary content type. It may be one of the following values:

Value	Description
MIME_CONTENT_UNKNOWN	The default content type for the specified extension should be used. This value should only be used to delete a previously defined content type.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

lpszSubtype

A null-terminated string which specifies the MIME subtype. This parameter may be NULL if the content type association is being deleted.

lpszExtension

A null-terminated string which specifies the file name extension that will be associated with the MIME content type.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeSetFileContentType** function allows an application to specify a content type for a given file extension, and is typically used to define custom content types for file attachments. The content type will override any default content types associated with the extension, as well as allow new content types to be defined for application-specific files.

Example

In the following example, the file extension ".dat" is associated with a custom content type, a file is attached to the message and then the custom content type is deleted. Note that because the primary content type designates the file as an application specific (non-text) file, it will be automatically encoded when attached to a message:

```
if (MimeSetFileContentType(hMessage, MIME_CONTENT_APPLICATION, "octet-stream",
"dat"))
{
    bResult = MimeAttachFile(hMessage, lpszFileName, MIME_ATTACH_DEFAULT);
    MimeSetFileContentType(hMessage, MIME_CONTENT_UNKNOWN, NULL, "dat");
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAttachData, MimeAttachFile, MimeGetFileContentType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetLastError Function

```
VOID WINAPI MimeSetLastError(
   DWORD dwErrorCode
);
```

The **MimeSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the caller. A value of zero clears the last error code.

Return Value

None.

Remarks

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those functions which call **MimeSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **MimeGetLastError** function. The use of **MimeGetLastError** is optional. An application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeGetErrorString, MimeGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetMessageDate Function

```
BOOL WINAPI MimeSetMessageDate(
    HMESSAGE hMessage,
    LPCTSTR lpszDate,
    BOOL bLocalize
);
```

The **MimeSetMessageDate** function sets the date and time in the header for the specified message.

Parameters

hMessage

Handle to the message.

lpszDate

Pointer to a string which specifies the date and time. If this parameter specifies a zero-length string or a NULL pointer, the current date and time will be used.

bLocalize

Boolean flag that specifies the date and time should be localized for the current timezone.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The date string should be in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **MimeFormatDate** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeFormatDate, MimeGetMessageDate, MimeGetMessageHeader, MimeParseDate, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetMessageHeader Function

```
BOOL WINAPI MimeSetMessageHeader(
    HMESSAGE hMessage,
    LPCTSTR lpszHeader,
    LPCTSTR lpszValue
);
```

The **MimeSetMessageHeader** function adds or updates a header field in the specified message.

Parameters

hMessage

Handle to the message.

lpszHeader

Pointer to a string which specifies the header field that will be added or updated.

lpszValue

Pointer to a string which specifies the value for the header field. This pointer may be NULL, which causes the header field to be removed from the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAddMessageHeaders, MimeEnumMessageHeaders, MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetMessagePart, MimeGetNextMessageHeader, MimeSetMessageHeaderEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetMessageHeaderEx Function

```
BOOL WINAPI MimeSetMessageHeaderEx(
    HMESSAGE hMessage,
    INT nMessagePart,
    LPCTSTR lpszHeader,
    LPCTSTR lpszValue,
    DWORD dwReserved
);
```

The **MimeSetMessageHeaderEx** function adds or updates a header field in the specified message.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value which specifies which part of the message the header should be set or modified in. A value of zero sets a header value in the main message header block, while a value greater than zero sets the header value in that specific part of a multipart message. A value of -1 specifies that the header value should be set in the current message part.

lpszHeader

Pointer to a string which specifies the header field that will be added or updated.

lpszValue

Pointer to a string which specifies the value for the header field. This pointer may be NULL, which causes the header field to be removed from the message.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAddMessageHeaders, MimeEnumMessageHeaders, MimeGetFirstMessageHeader, MimeGetMessageHeader, MimeGetMessagePart, MimeGetNextMessageHeader, MimeSetMessagePart, MimeSetMessageText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetMessagePart Function

```
INT WINAPI MimeSetMessagePart(
   HMESSAGE hMessage,
   INT nNewPart
);
```

The **MimeSetMessagePart** function sets the current message part index for the specified message.

Parameters

hMessage

Handle to the message.

nNewPart

The new message part index. A value of zero specifies the main message part.

Return Value

If the function succeeds, the return value is the previous message part index. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

Multipart email messages are structured using the MIME (Multipurpose Internet Mail Extensions) standard, which allows a single email to contain multiple content types, such as plain text, HTML, and file attachments, within a single message body. These components are organized as discrete parts, each with its own headers and body, separated by unique MIME boundaries. In a typical `multipart/alternative` message, part 0 often contains the message preamble (e.g.: "This is a multipart message in MIME format"), part 1 may contain the plain text version of the message, and part 2 may contain the HTML version. However, multipart messages are not required to include both plain text and HTML content. A message may only include an HTML part and one or more attachments, or any other combination of content types. The structure and ordering of parts depends entirely on how the original sender or mail client composed the message.

To simplify navigating these multipart structures, the **MimeSetMessagePart** and **MimeGetMessagePart** functions enable you to iterate through the individual parts of a message. The message parts are zero-based, with index 0 always referring to the top-level message headers and preamble. By calling **MimeSetMessagePart**, you can change the currently selected message part, enabling functions like **MimeGetMessageHeader** or **MimeGetMessageText** to operate on that specific segment. Calling **MimeGetMessagePart** returns the current part index, which can be useful for saving and restoring state while iterating over a message.

The internal MIME parser flattens multipart messages, meaning that if a message contains nested multipart sections (such as a multipart/mixed part containing a multipart/alternative part), all leaf parts are presented in a linear sequence. This simplifies access and avoids the complexity of recursive traversal. You can determine the total number of parts using **MimeGetMessagePartCount**, and then iterate from part 0 to part N-1 to examine or extract each segment in order. This approach provides both fine-grained access to MIME message parts and an intuitive model for parsing even complex message structures.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csmsgv12.lib

See Also

MimeGetMessagePart, MimeGetMessagePartCount, MimeGetMessageHeader, MimeGetMessageText, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetMessageText Function

```
LONG WINAPI MimeSetMessageText(
    HMESSAGE hMessage,
    LONG nOffset,
    LPCTSTR lpszText
);
```

The **MimeSetMessageText** function copies the specified text into the body of the current message part.

Parameters

hMessage

Handle to the message.

nOffset

The offset into the body of the message part. A value of -1 specifies that the text will be appended to the message body.

lpszText

A null-terminated string which specifies the text to be copied to the current message part at the given offset.

Return Value

If the function succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be copied into the current message part. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeAppendMessageText, MimeClearMessageText, MimeCompareMessageText, MimeGetMessageText, MimeResetMessage, MimeSetMessageHeader, MimeSetMessagePart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeSetMessageVersion Function

```
BOOL WINAPI MimeSetMessageVersion(
    HMESSAGE hMessage,
    LPCTSTR lpszVersion
);
```

Parameters

hMessage

Handle to the message.

lpszVersion

Pointer to a string which specifies the MIME version. A value of NULL sets the version to the default value of 1.0.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

MimeGetMessageVersion, MimeSetMessageHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeStoreMessage Function

```
LONG WINAPI MimeStoreMessage(
   HMESSAGESTORE hStorage,
   HMESSAGE hMessage,
   DWORD dwReserved
);
```

The **MimeStoreMessage** function stores the specified message in a message store.

Parameters

hStorage

Handle to the message store.

hMessage

Handle to the message that will be stored.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is the message number for the message that was just stored. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeStoreMessage** function will always append the specified message to the storage file. If you want to replace a message in the message store, you should use the **MimeReplaceStoredMessage** function.

Example

```
HMESSAGE hMessage;

// Compose a new message
hMessage = MimeComposeMessage(lpszSender,
                              lpszRecipient,
                              NULL,
                              lpszSubject,
                              lpszMessage,
                              NULL,
                              MIME_CHARSET_DEFAULT,
                              MIME_ENCODING_DEFAULT);

if (hMessage != INVALID_MESSAGE)
{
    HMESSAGESTORE hStorage;

    // Open the message storage file
    hStorage = MimeOpenMessageStore(lpszFileName, MIME_STORAGE_WRITE);

    if (hStorage == INVALID_MESSAGESTORE)
    {
        // Close the message and return if we are unable to
        // open the storage file
```

```
        MimeCloseMessage(hMessage);
        return;
    }

    // Store a copy of the message in the message store
    nMessageId = MimeStoreMessage(hStorage, hMessage, 0);

    if (nMessageId == MIME_ERROR)
    {
        // We were unable to store the message
    }

    // Close the message store
    MimeCloseMessageStore(hStorage);

    // Close the message that was created
    MimeCloseMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeDeleteStoredMessage, MimeFindStoredMessage, MimeGetStoredMessage, MimeReplaceStoredMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

MimeUninitialize Function

```
VOID WINAPI MimeUninitialize();
```

The **MimeUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **MimeInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **MimeUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **MimeUninitialize** for every successful call to **MimeInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmsgv12.lib

See Also

MimeInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Message Character Sets

Value	Name	Code Page	Description
MIME_CHARSET_USASCII	us-ascii	20127	A character set which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered.
MIME_CHARSET_ISO8859_1	iso-8859-1	28591	A character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin-1. This character set is similar to Windows code page 1252 (Windows-1252), however there are differences such as the Euro symbol.
MIME_CHARSET_ISO8859_2	iso-8859-2	28592	A character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin-2. This character set is similar to Windows code page 1250, however the characters are arranged differently.
MIME_CHARSET_ISO8859_3	iso-8859-3	28593	A character set for southern European languages such as Maltese and Esperanto. This character set was also used with the Turkish language, but it was superseded by ISO 8859-9 which is the preferred character set for Turkish. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_4	iso-8859-4	28594	A character set for northern European languages such as Latvian, Lithuanian and Greenlandic. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_5	iso-8859-5	28595	A character set for Cyrillic languages such as Russian, Bulgarian and Serbian. This character set was never widely adopted and most mail messages use

			either KOI8 or UTF-8 encoding.
MIME_CHARSET_ISO8859_6	iso-8859-6	28596	A character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	iso-8859-7	28597	A character set for the Greek language. This character set is also commonly referred to as Latin/Greek. This character set is no longer widely used and has largely been replaced with UTF-8 which provides more complete coverage of the Greek alphabet.
MIME_CHARSET_ISO8859_8	iso-8859-8	28598	A character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. In most cases it is recommended that you use UTF-8 instead of this character set.
MIME_CHARSET_ISO8859_9	iso-8859-9	28599	A character set for the Turkish language. This character set is also commonly referred to as Latin-5. This character set is nearly identical to ISO 8859-1, except that it replaces certain Icelandic characters with Turkish characters.
MIME_CHARSET_ISO8859_10	iso-8859-10	28600	A character set for the Danish, Icelandic, Norwegian and Swedish languages. This character set is also commonly referred to as Latin-6 and is similar to ISO 8859-4.
MIME_CHARSET_ISO8859_13	iso-8859-13	28603	A character set for Baltic languages. This character set is also commonly referred to as Latin-7. This character set is similar to ISO 8859-4, except it adds certain Polish characters and does not support Nordic languages.
MIME_CHARSET_ISO8859_14	iso-8859-14	28604	A character set for Gaelic languages such as Irish, Manx and Scottish Gaelic. This character set is also commonly

			referred to as Latin-8. This character set replaced ISO 8859-12 which was never fully implemented.
MIME_CHARSET_ISO8859_15	iso-8859-15	28605	A character set for western European languages. This character set is also commonly referred to as Latin-9 and is nearly identical to ISO8859-1 except that it replaces lesser-used symbols with the Euro sign and some letters.
MIME_CHARSET_ISO2022_JP	iso-2022-jp	50220	A multi-byte character encoding for Japanese that is widely used with mail messages. This is a 7-bit encoding where all characters start with ASCII and uses escape sequences to switch to the double-byte character sets.
MIME_CHARSET_ISO2022_KR	iso-2022-kr	50225	A multi-byte character encoding for Korean which encodes both ASCII and Korean double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_ISO2022_CN	x-cp50227	50227	A multi-byte character encoding for Simplified Chinese which encodes both ASCII and Chinese double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_KOI8R	koi8-r	20866	A character set for Russian using the Cyrillic alphabet. This character set also covers the Bulgarian language. Most mail messages in the Russian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_KOI8U	koi8-u	21866	A character set for Ukrainian using the Cyrillic alphabet. This character set is similar to the KOI8-R character set, but replaces certain symbols with Ukrainian letters. Most mail messages in the Ukrainian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_GB2312	x-cp20936	20936	A multi-byte character encoding which can represent ASCII and simplified Chinese characters. It has been superseded by GB18030, however it

			remains widely used in China.
MIME_CHARSET_GB18030	gb18030	54936	A Unicode transformation format which can represent all Unicode code points and supports both simplified and traditional Chinese characters. It is backwards compatible with GB2312 and supersedes that character set.
MIME_CHARSET_BIG5	big5	950	A multi-byte character set that supports both ASCII characters and traditional Chinese characters. It is widely used in Taiwan, Hong Kong and Macau. It is no longer commonly used in China, which has developed GB18030 as a standard encoding. Microsoft's implementation of Big5 on Windows does not support all of the extensions and is missing certain code points.
MIME_CHARSET_UTF7	utf-7	65000	A Unicode transformation format that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 when quoted-printable or base64 encoding is not desired.
MIME_CHARSET_UTF8	utf-8	65001	A Unicode transformation format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it is recommended that you use either quoted-printable or base64 encoding to ensure compatibility with mail servers that do not support 8-bit characters.
MIME_CHARSET_UTF16	utf-16le	N/A	A 16-bit Unicode format that represents each character as a 16-bit value in little endian byte order. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead. UTF-16 characters in big endian byte order are not supported.

Remarks

When composing a new message, it is recommended that you always use UTF-8 as the character set encoding which ensures broad compatibility with most applications. The other character sets are primarily used when parsing messages generated by other applications. Internally, all message headers and text are processed as UTF-8. If you use the ANSI version of these functions, header values

and message text will always be returned to your application as UTF-8 encoded Unicode, regardless of the original character set used in the message.

In addition to the character sets listed above, the library will recognize additional character sets which correspond to specific Windows code pages, as well several variants. These additional character sets are included for compatibility with other applications; they are not defined because they should not be used when composing new messages.

It is important to note that while certain Windows character sets are similar to standard ISO character sets, they are not identical. For example, although the Windows-1252 character set is nearly identical to ISO 8859-1, they are not interchangeable. Some legacy applications make the error of representing Windows ANSI character sets as 8-bit ISO character sets, which can result in errors when converting them to Unicode. This is something to be aware of when encoding and decoding text generated by older applications. Before the widespread adoption of UTF-8, it was particularly common for legacy Windows mail clients to default to using Windows-1252 for text and label it as using ISO 8859-1.

Although the library supports UTF-16, it is recommended you use UTF-8 instead. Text which uses UTF-16 will always be base64 encoded, and some mail clients may not recognize it as a valid character set. If the message does not specify if big endian or little endian byte order is used, the library will default to little endian. When UTF-16 is used when composing a new message, it will always use little endian byte order.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

MimeComposeMessage, MimeCreateMessagePartEx, MimeDecodeTextEx, MimeEncodeTextEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol Library

Download and submit articles to a news server.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSNWSV12.DLL
Version	12.0.3024.1286
LibID	C11CFCC7-8D02-4CB5-B3D2-003BD1D65C40
Import Library	CSNWSV12.LIB
Dependencies	None
Standards	RFC 977, RFC 2980

Overview

The Network News Transfer Protocol (NNTP) is used with servers that provide news services. This is similar in functionality to bulletin boards or message boards, where topics are organized hierarchically into groups, called newsgroups. Users can browse and search for messages, called news articles, which have been posted by other users. On many servers, they can also post their own articles which can be read by others. The largest collection of public newsgroups available is called USENET, a world-wide distributed discussion system. In addition, there are a large number of smaller news servers. For example, Catalyst Development operates a news server which functions as a forum for technical questions and announcements.

The SocketTools library provides a comprehensive interface for accessing newsgroups, retrieving articles and posting new articles. In combination with the Mail Message library to process the news articles, SocketTools can be used to integrate newsgroup access with an existing email application, or you can implement your own full-featured newsgroup client.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of

Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol Functions

Function	Description
NntpAsyncConnect	Establish an asynchronous connection to the specified server
NntpAttachThread	Attach the specified client handle to another thread
NntpAuthenticate	Authenticate the specified user on the news server
NntpCancel	Cancel the current blocking operation
NntpCloseArticle	Close the article being posted to the current newsgroup
NntpCommand	Send a command to the server
NntpConnect	Connect to the specified server
NntpCreateArticle	Create a new article in the current newsgroup
NntpCreateSecurityCredentials	Create a new security credentials structure
NntpDeleteSecurityCredentials	Delete a previously created security credentials structure
NntpDisableEvents	Disable asynchronous event notification
NntpDisableTrace	Disable logging of network function calls
NntpDisconnect	Disconnect from the current server
NntpEnableEvents	Enable asynchronous event notification
NntpEnableTrace	Enable logging of network function calls to a file
NntpEventProc	Callback function that processes events generated by the client
NntpFreezeEvents	Suspend asynchronous event processing
NntpGetArticle	Retrieve an article from the server and store the contents in a local buffer
NntpGetArticleEx	Retrieve an article from the server with support for large article IDs
NntpGetArticleByMessageId	Retrieve an article from the server using a message ID and store the contents in a local buffer
NntpGetArticleHeaders	Return the contents of the specified article header
NntpGetArticleHeadersEx	Return the contents of the specified article header with support for large article IDs
NntpGetArticleMessageId	Return the message identifier for the specified article
NntpGetArticleMessageIdEx	Return the message identifier for the specified article with support for large article IDs
NntpGetArticleRange	Return the first and last article number for the current group
NntpGetArticleRangeEx	Return the first and last article number for the current group with support for large article IDs
NntpGetArticleSize	Return the size of the specified news article in bytes
NntpGetArticleSizeEx	Return the size of the specified article with support for large article IDs
NntpGetCurrentArticle	Return the current article number for the selected group
NntpGetCurrentArticleEx	Return the current article number for the selected group with support for large article IDs
NntpGetCurrentDate	Return the current date and time
NntpGetErrorString	Return a description for the specified error code
NntpGetFirstArticle	Return the first available article in the currently selected newsgroup
NntpGetFirstArticleEx	Return the first available article in the currently selected newsgroup with support for large article IDs
NntpGetFirstGroup	Return the first available newsgroup from the server
NntpGetFirstGroupEx	Return the first available newsgroup from the server with support for large article IDs
NntpGetGroupName	Return the name of the currently selected newsgroup
NntpGetGroupTitle	Return a description of the currently selected newsgroup

NntpGetLastError	Return the last error code
NntpGetMessageIdArticle	Return the article number for the specified message identifier
NntpGetMessageIdArticleEx	Return the article number for the specified message with support for large article IDs
NntpGetMultiLine	Return the client multi-line output flag
NntpGetNextArticle	Return the next available article from the current newsgroup
NntpGetNextArticleEx	Return the next available article from the current newsgroup with support for large article IDs
NntpGetNextGroup	Return the next available newsgroup from the server
NntpGetNextGroupEx	Return the next available newsgroup from the server with support for large article IDs
NntpGetResultCode	Return the result code from the previous command
NntpGetResultString	Return the result string from the previous command
NntpGetSecurityInformation	Return security information about the current client connection
NntpGetStatus	Return the current client status
NntpGetTimeout	Return the number of seconds until an operation times out
NntpGetTransferStatus	Return data transfer statistics
NntpGetTransferStatusEx	Return data transfer statistics with support for large article IDs
NntpInitialize	Initialize the library and validates the specified license key at runtime
NntpIsBlocking	Determine if the client is blocked, waiting for information
NntpIsConnected	Determine if the client is connected to the server
NntpIsReadable	Determine if data can be read from the server
NntpIsWritable	Determine if data can be written to the server
NntpListArticles	Return a list of articles in the currently selected newsgroup
NntpListArticlesEx	Return a list of articles in the currently selected newsgroup with support for large article IDs
NntpListGroups	Return a list of newsgroups maintained by the server
NntpListNewGroups	Return a list of newsgroups created since a specified date
NntpOpenArticle	Open the specified article in the current newsgroup
NntpOpenArticleEx	Open the specified article in the current newsgroup with support for large article IDs
NntpOpenArticleByMessageId	Open the specified article by message ID in the current newsgroup
NntpOpenNextArticle	Open the next available article
NntpOpenPreviousArticle	Open the previous article
NntpPostArticle	Post a new article to the news server
NntpRead	Read data returned by the news server
NntpRegisterEvent	Register an event callback function
NntpReset	Reset the client
NntpSelectGroup	Select the specified newsgroup to retrieve articles from
NntpSetLastError	Set the last error code
NntpSetMultiLine	Set the client multi-line output flag
NntpSetTimeout	Set the number of seconds until an operation times out
NntpStoreArticle	Retrieve an article and store the contents in a local file
NntpStoreArticleEx	Retrieve an article and store the contents in a local file with support for large article IDs
NntpUninitialize	Terminate use of the library by the application
NntpWrite	Write data to the news server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpAsyncConnect Function

```
HCLIENT WINAPI NntpAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **NntpAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **NntpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
NNTP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
NNTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
NNTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection

	will always be permitted. This option only affects connections using the TLS protocol.
NNTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
NNTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
NNTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
NNTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the NNTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **NntpGetLastError**.

Remarks

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle.

The ownership of the handle may be transferred from one thread to another using the **NntpAttachThread** function.

Specifying the NNTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_LISTARTICLE	The client is retrieving a list of articles from the server and information about another article has been processed. The client should use this event to retrieve the next available article from the server.
NNTP_EVENT_LASTARTICLE	The client is retrieving a list of articles from the server and all of the available articles have been processed. This event is generated when there are no further articles.
NNTP_EVENT_LISTGROUP	The client is retrieving a list of newsgroups from the server and

	information about another group has been processed. The client should use this event to retrieve the next available newsgroup from the server.
NNTP_EVENT_LASTGROUP	The client is retrieving a list of newsgroups from the server and all of the available groups have been processed. This event is generated when there are no further newsgroups.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

To cancel asynchronous notification and return the client to a blocking mode, use the **NntpDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpAuthenticate, NntpConnect, NntpCreateSecurityCredentials, NntpDeleteSecurityCredentials, NntpDisconnect, NntpInitialize, NntpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpAttachThread Function

```
DWORD WINAPI NntpAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **NntpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **NntpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **NntpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **NntpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **NntpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpConnect, NntpDisconnect, NntpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpAuthenticate Function

```
INT WINAPI NntpAuthenticate(
   HCLIENT hClient,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **NntpAuthenticate** function is used to authenticate access to the news server. Not all news servers require authentication by the client.

Parameters

hClient

Handle to the client session.

lpszUserName

Pointer to a string which specifies the user name required for authentication on the news server.

lpszPassword

Pointer to a string which specifies the password required for authentication on the news server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function should only be called if the server requires authentication. Two authentication methods, "original" and "simple" authentication, are recognized by the library.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpConnect, NntpListArticles, NntpListGroups

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpCancel Function

```
INT WINAPI NntpCancel(
   HCLIENT hClient
);
```

The **NntpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When the **NntpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpIsBlocking, NntpRead, NntpReset, NntpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpCloseArticle Function

```
INT WINAPI NntpCloseArticle(
    HCLIENT hClient
);
```

The **NntpCloseArticle** function closes the current article that has been opened or created.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

If an article is being created, this function actually submits the article to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCreateArticle, NntpOpenArticle, NntpOpenArticleByMessageId, NntpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpCommand Function

```
INT WINAPI NntpCommand(
   HCLIENT hClient,
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **NntpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the function succeeds, the return value is the result code returned by the server. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetMultiLine, NntpGetResultCode, NntpGetResultString, NntpSetMultiLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpConnect Function

```
HCLIENT WINAPI NntpConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **NntpConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
NNTP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
NNTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
NNTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
NNTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
NNTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will

	allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
NNTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
NNTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the NNTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **NntpGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **NntpConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **NntpAttachThread** function.

Specifying the NNTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpAuthenticate, NntpCreateSecurityCredentials, NntpDeleteSecurityCredentials, NntpDisconnect, NntpInitialize, NntpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpCreateArticle Function

```
INT WINAPI NntpCreateArticle(
    HCLIENT hClient
);
```

The **NntpCreateArticle** function creates a new article in the current newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function sends the POST command to the news server. Not all servers permit clients to post articles. The client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

The **NntpCloseArticle** function must be called once the contents of the article has been written to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCloseArticle, NntpGetCurrentDate, NntpListArticles, NntpOpenArticle, NntpOpenArticleByMessageId, NntpOpenNextArticle, NntpOpenPreviousArticle, NntpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpCreateSecurityCredentials Function

```
BOOL WINAPI NntpCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **NntpCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **NntpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **NntpGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **NntpConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
NntpCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                              0,
                              NULL,
                              NULL,
                              lpszCertStore,
                              lpszCertName,
                              NULL,
                              &lpSecCred);

hClient = NntpConnect(lpszHostName,
                      NNTP_PORT_SECURE,
                      NNTP_TIMEOUT,
                      NNTP_OPTION_SECURE,
                      lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpConnect, NntpDeleteSecurityCredentials, NntpGetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpDeleteSecurityCredentials Function

```
VOID WINAPI NntpDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **NntpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)
    NntpDeleteSecurityCredentials(&lpSecCred);

NntpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpCreateSecurityCredentials, NntpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpDisableEvents Function

```
INT WINAPI NntpDisableEvents(
    HCLIENT hClient
);
```

The **NntpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpEnableEvents, NntpFreezeEvents, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpDisableTrace Function

```
BOOL WINAPI NntpDisableTrace();
```

The **NntpDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpDisconnect Function

```
INT WINAPI NntpDisconnect(
   HCLIENT hClient
);
```

The **NntpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpConnect, NntpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpEnableEvents Function

```
INT WINAPI NntpEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **NntpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **NntpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the ***hEventWnd*** parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_LISTARTICLE	The client is retrieving a list of articles from the server and information about another article has been processed. The client should use this event to retrieve the next available article from the server.
NNTP_EVENT_LASTARTICLE	The client is retrieving a list of articles from the server and all of the available articles have been processed. This event is generated when there are no further articles.
NNTP_EVENT_LISTGROUP	The client is retrieving a list of newsgroups from the server and information about another group has been processed. The client should use this event to retrieve the next available newsgroup from the server.
NNTP_EVENT_LASTGROUP	The client is retrieving a list of newsgroups from the server and all of the available groups have been processed. This event is generated when there are no further newsgroups.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpDisableEvents, NntpFreezeEvents, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpEnableTrace Function

```
BOOL WINAPI NntpEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **NntpEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpEventProc Function

```
VOID CALLBACK NntpEventProc(
   HCLIENT hClient,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **NntpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **NntpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **NntpRegisterEvent** function. The **NntpEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpDisableEvents, NntpEnableEvents, NntpFreezeEvents, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpFreezeEvents Function

```
INT WINAPI NntpFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **NntpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpDisableEvents, NntpEnableEvents, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticle Function

```
INT WINAPI NntpGetArticle(
   HCLIENT hClient,
   LONG nArticleId,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved
);
```

The **NntpGetArticle** function retrieves the specified article and copies the contents to a local buffer.

Parameters

hClient

Handle to the client session.

nArticleId

Number of article to retrieve from the server. This value must be greater than zero.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticle** function is used to retrieve an article from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the article. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the article data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the complete article has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback

function using the **NntpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **NntpGetTransferStatus** function. If you need to retrieve an article based on its message ID rather than the article number, use the **NntpGetArticleByMessageId** function.

Your application should use the **NntpGetArticleEx** function if the server uses 64-bit article IDs.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the article into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = NntpGetArticle(hClient,
                         nArticleId,
                         &hgblBuffer,
                         &cbBuffer,
                         0);

if (nResult != NNTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // article text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCreateArticle, NntpEnableEvents, NntpGetArticleEx, NntpGetArticleByMessageId, NntpGetArticleHeaders, NntpGetTransferStatus, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleByMessageId Function

```
INT WINAPI NntpGetArticleByMessageId(
    HCLIENT hClient,
    LPCTSTR lpszMessageId,
    LPVOID lpvBuffer,
    LPDWORD lpdwLength,
    DWORD dwReserved
);
```

The **NntpGetArticleByMessageId** function retrieves an article using the specified message ID and copies the contents to a local buffer.

Parameters

hClient

Handle to the client session.

lpszMessageId

A null-terminated string which specifies the message ID of the article that you want to retrieve. If this parameter is a NULL pointer or specifies a zero-length string, an error will be returned.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleByMessageId** function is used to retrieve an article from the server and copy it into a local buffer. This function is identical to the **NntpGetArticle** function except that a message ID string is used to identify the article, rather than an article number. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the article. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the article data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the complete article has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **NntpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the article into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = NntpGetArticleByMessageId(hClient,
                                    lpszMessageId,
                                    &hgblBuffer,
                                    &cbBuffer,
                                    0);

if (nResult != NNTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // article text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpCreateArticle, NntpEnableEvents, NntpGetArticle, NntpGetArticleHeaders, NntpGetTransferStatus, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleHeaders Function

```
INT WINAPI NntpGetArticleHeaders(
    HCLIENT hClient,
    LONG nArticleId,
    LPVOID lpvHeaders,
    LPDWORD lpdwLength
);
```

The **NntpGetArticleHeaders** function retrieves the headers for the specified article from the server.

Parameters

hClient

Handle to the client session.

nArticleId

Number of article to retrieve from the server. This value must be greater than zero.

lpvHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvHeaders*** parameter. If the ***lpvHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleHeaders** function is used to retrieve an article header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpvHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvHeaders*** parameter point to a global memory handle which will contain the message headers when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

Your application should use the **NntpGetArticleHeadersEx** function if the server uses 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetArticle, NntpGetArticleHeadersEx, NntpGetArticleRange, NntpListArticles, NntpPostArticle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleHeadersEx Function

```
INT WINAPI NntpGetArticleHeadersEx(
    HCLIENT hClient,
    ULONGLONG nArticleId,
    LPVOID lpvHeaders,
    LPDWORD lpdwLength
);
```

The **NntpGetArticleHeadersEx** function retrieves the headers for the specified article from the server.

Parameters

hClient

Handle to the client session.

nArticleId

Number of article to retrieve from the server.

lpvHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvHeaders*** parameter. If the ***lpvHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleHeadersEx** function is used to retrieve an article header block from the server and copy it into a local buffer. This function provides support for servers which use 64-bit article IDs. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpvHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvHeaders*** parameter point to a global memory handle which will contain the message headers when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetArticle, NntpGetArticleRangeEx, NntpListArticlesEx, NntpPostArticle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleMessageId Function

```
INT WINAPI NntpGetArticleMessageId(
    HCLIENT hClient,
    LONG nArticleId,
    LPTSTR lpszMessageId,
    INT cbMessageId
);
```

The **NntpGetArticleMessageId** function returns the message identifier for the specified article in the current newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

Article number to retrieve the message identifier for. The value may be zero, in which case the current article number is used.

lpszMessageId

Pointer to a string buffer which will contain the message identifier for the specified article.

cbMessageId

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the message identifier string. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The message identifier is a string which can uniquely identify the message on the news server. This value may be used to retrieve the contents of the article. Your application should use the **NntpGetArticleMessageIdEx** function if the server uses 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticleMessageIdEx, NntpGetMessageIdArticle, NntpListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleMessageIdEx Function

```
INT WINAPI NntpGetArticleMessageIdEx(
    HCLIENT hClient,
    ULONGLONG nArticleId,
    LPTSTR lpszMessageId,
    INT cbMessageId
);
```

The **NntpGetArticleMessageIdEx** function returns the message identifier for the specified article in the current newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

Article number to retrieve the message identifier for. The value may be zero, in which case the current article number is used.

lpszMessageId

Pointer to a string buffer which will contain the message identifier for the specified article.

cbMessageId

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the message identifier string. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The message identifier is a string which can uniquely identify the message on the news server. This value may be used to retrieve the contents of the article. This function provides support for servers which use 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetMessageIdArticleEx, NntpListArticlesEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleRange Function

```
LONG WINAPI NntpGetArticleRange(
   HCLIENT hClient,
   LPLONG lpnFirstArticle,
   LPLONG lpnLastArticle
);
```

The **NntpGetArticleRange** function returns the first and last article numbers for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpnFirstArticle

Pointer to a long integer that will contain the first article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnLastArticle

Pointer to a long integer that will contain the last article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

Return Value

If the function succeeds, the return value is number of articles in the selected newsgroup. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. If the server uses 64-bit article IDs, your application should use the **NntpGetArticleRangeEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetArticleRangeEx, NntpOpenArticle, NntpOpenArticleByMessageId, NntpListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleRangeEx Function

```
BOOL WINAPI NntpGetArticleRangeEx(
   HCLIENT hClient,
   ULONGLONG *lpnFirstArticle,
   ULONGLONG *lpnLastArticle,
   ULONGLONG *lpnArticleCount
);
```

The **NntpGetArticleRangeEx** function returns the first and last article numbers for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpnFirstArticle

Pointer to an unsigned 64-bit integer that will contain the first article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnLastArticle

Pointer to an unsigned 64-bit integer that will contain the last article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnArticleCount

Pointer to an unsigned 64-bit integer that will contain the total number of articles in the selected newsgroup. If this parameter is NULL, it will be ignored.

Return Value

If the function succeeds, the return value will be non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. This function provides support for servers which use 64-bit article IDs.

The **NntpGetArticleRangeEx** function returns a BOOL (a signed 32-bit integer) to indicate success or failure, and the article count is returned in a variable that is passed by reference to the function. If you are updating your code to use this function, make sure you also change how you check the return value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpOpenArticleEx, NntpOpenArticleByMessageId, NntpListArticlesEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleSize Function

```
DWORD WINAPI NntpGetArticleSize(
    HCLIENT hClient,
    LONG nArticleId
);
```

The **NntpGetArticleSize** function returns the size of the specified article.

Parameters

hClient

Handle to the client session.

nArticleId

An integer value that identifies the article. The value may be zero, in which case the current article number is used.

Return Value

If the function succeeds, the return value is the size of the article in bytes. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleSize** function sends the XOVER command to the server to request the size of the specified article. If the server uses 64-bit article IDs, your application should use the **NntpGetArticleSizeEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetArticleSizeEx, NntpOpenArticle, NntpOpenArticleByMessageId, NntpListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetArticleSizeEx Function

```
DWORD WINAPI NntpGetArticleSizeEx(
    HCLIENT hClient,
    ULONGLONG nArticleId
);
```

The **NntpGetArticleSize** function returns the size of the specified article.

Parameters

hClient

Handle to the client session.

nArticleId

An unsigned 64-bit integer value that identifies the article. The value may be zero, in which case the current article number is used.

Return Value

If the function succeeds, the return value is the size of the article in bytes. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleSizeEx** function sends the XOVER command to the server to request the size of the specified article. This function provides support for servers which use 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpOpenArticleEx, NntpOpenArticleByMessageId, NntpListArticlesEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetCurrentArticle Function

```
LONG WINAPI NntpGetCurrentArticle(
    HCLIENT hClient
);
```

The **NntpGetCurrentArticle** function returns the current article number for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current article number. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpOpenArticle, NntpOpenArticleByMessageId, NntpGetArticleRange, NntpGetFirstArticle, NntpGetNextArticle, NntpListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetCurrentArticleEx Function

```
BOOL WINAPI NntpGetCurrentArticleEx(
    HCLIENT hClient,
    ULONGLONG *lpnArticleId
);
```

The **NntpGetCurrentArticle** function returns the current article number for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpnArticleId

Pointer to an unsigned 64-bit integer that will contain the current article number in the selected newsgroup. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetCurrentArticleEx** function returns a BOOL (a signed 32-bit integer) to indicate success or failure, and the article ID is returned in a variable that is passed by reference to the function. This function provides support for servers which use 64-bit article IDs. If you are updating your code to use this function, make sure you also change how you check the return value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpOpenArticleEx, NntpOpenArticleByMessageId, NntpGetArticleRangeEx, NntpGetFirstArticleEx, NntpGetNextArticleEx, NntpListArticlesEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetCurrentDate Function

```
INT WINAPI NntpGetCurrentDate(
    LPTSTR lpszDate,
    INT nMaxLength
);
```

The **NntpGetCurrentDate** function copies the current date and time to the specified buffer in a format that is commonly used in news articles. This date format should be used in all date-related fields in the message header.

Parameters

lpszDate

Pointer to a string buffer that will contain the current date and time when the function returns.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpCreateArticle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetErrorString Function

```
INT WINAPI NntpGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **NntpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **NntpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetLastError, NntpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetFirstArticle Function

```
BOOL WINAPI NntpGetFirstArticle(
   HCLIENT hClient,
   LPNEWSARTICLE lpArticle
);
```

The **NntpGetFirstArticle** function returns information about the first article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticle

A pointer to a NEWSARTICLE structure which will contain information about the first article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstArticle** function returns information about the first article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetNextArticle** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticle** function cannot be called while inside a loop calling **NntpGetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NttpGetArticle** for each article number to retrieve the article text.

The date and time that the article was posted is returned in the ***stPosted*** member of the NEWSARTICLE structure. This value is returned in Universal Coordinated Time (UTC) and can be converted to local time using the **SystemTimeToTzSpecificLocalTime** function.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstArticleEx** and **NntpGetNextArticleEx** functions.

Example

```
NEWSARTICLE newsArticle;
BOOL bResult;
INT nResult;

// List all articles in the current group
nResult = NntpListArticles(hClient, -1L, -1L);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();
```

```
    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}

// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetFirstArticle(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticle(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticle, NntpGetFirstArticleEx, NntpGetFirstGroup, NntpGetNextArticle, NntpGetNextGroup, NntpGetNextGroupEx, NntpListArticles, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetFirstArticleEx Function

```
BOOL WINAPI NntpGetFirstArticleEx(
   HCLIENT hClient,
   LPNEWSARTICLEEX lpArticleEx
);
```

The **NntpGetFirstArticleEx** function returns information about the first article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticleEx

A pointer to a NEWSARTICLEEX structure which will contain information about the first article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstArticleEx** function returns information about the first article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetNextArticleEx** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access. This function provides support for servers which use 64-bit article IDs.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticleEx** function cannot be called while inside a loop calling **NntpGetNextArticleEx**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NttpGetArticleEx** for each article number to retrieve the article text.

The date and time that the article was posted is returned in the ***stPosted*** member of the NEWSARTICLE structure. This value is returned in Universal Coordinated Time (UTC) and can be converted to local time using the **SystemTimeToTzSpecificLocalTime** function.

Example

```
NEWSARTICLEEX newsArticle;
BOOL bResult;
INT nResult;

// List all articles in the current group
nResult = NntpListArticlesEx(hClient, (ULONGLONG)-1, (ULONGLONG)-1);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
```

```
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}

// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetFirstArticleEx(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%I64u %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticleEx(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticleEx, NntpGetFirstGroupEx, NntpGetNextArticleEx, NntpGetNextGroupEx, NntpListArticlesEx, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetFirstGroup Function

```
BOOL WINAPI NntpGetFirstGroup(
   HCLIENT hClient,
   LPNEWSGROUP lpGroup
);
```

The **NntpGetFirstGroup** function returns information about the first available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroup

A pointer to a NEWSGROUP structure which will contain information about the first available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no newsgroups available, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstGroup** function returns information about the first newsgroup on the server. This function is used in conjunction with the **NntpGetNextGroup** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstGroupEx** and **NntpGetNextGroupEx** functions.

Example

```
NEWSGROUP newsGroup;
BOOL bResult;
INT nResult;

// List all available newsgroups
nResult = NntpListGroups(hClient);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}
```

```
// Get each newsgroup, printing the article range and
// the name of the group
bResult = NntpGetFirstGroup(hClient, &newsGroup);

while (bResult)
{
    _tprintf(_T("%ld %ld %s\n"), newsGroup.nFirstArticle,
                                 newsGroup.nLastArticle,
                                 newsGroup.szName);

    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstArticle, NntpGetFirstGroupEx, NntpGetNextArticle, NntpGetNextGroup, NntpGetNextGroupEx, NntpListArticles, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetFirstGroupEx Function

```
BOOL WINAPI NntpGetFirstGroupEx(
   HCLIENT hClient,
   LPNEWSGROUPEX lpGroupEx
);
```

The **NntpGetFirstGroupEx** function returns information about the first available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroupEx

A pointer to a NEWSGROUPEX structure which will contain information about the first available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no newsgroups available, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstGroupEx** function returns information about the first newsgroup on the server. This function is used in conjunction with the **NntpGetNextGroupEx** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. This function provides support for servers which use 64-bit article IDs.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

Example

```
NEWSGROUPEX newsGroup;
BOOL bResult;
INT nResult;

// List all available newsgroups
nResult = NntpListGroups(hClient);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();
    NntpGetErrorString(dwError, szError, ASTRING);
    fprintf(stderr, "Error %08x: %s\n", dwError, szError);
    return;
}

// Get each newsgroup, printing the article range and
// the name of the group
```

```
bResult = NntpGetFirstGroupEx(hClient, &newsGroup);
while (bResult)
{
    printf("%I64u %I64u %s\n", newsGroup.nFirstArticle,
                               newsGroup.nLastArticle,
                               newsGroup.szName);

    bResult = NntpGetNextGroupEx(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstArticleEx, NntpGetNextArticleEx, NntpGetNextGroupEx, NntpListArticlesEx, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetGroupName Function

```
INT WINAPI NntpGetGroupName(
   HCLIENT hClient,
   LPTSTR lpszGroupName,
   INT nMaxLength
);
```

The **NntpGetGroupName** function returns the name of the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpszGroupName

Pointer to a string buffer that will contain the name of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the newsgroup name. If no newsgroup has been selected, the function will return a value of zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetGroupTitle, NntpListGroups, NntpListNewGroups, NntpSelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetGroupTitle Function

```
INT WINAPI NntpGetGroupTitle(
   HCLIENT hClient,
   LPTSTR lpszGroupTitle,
   INT nMaxLength
);
```

The **NntpGetGroupTitle** function returns a description of the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpszGroupTitle

Pointer to a string buffer that will contain a description of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the description. If no newsgroup has been selected, the function will return a value of zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The news server must support the XGTITLE command so that the group description can be obtained when the newsgroup is selected. If this command is not recognized, then no description will be returned.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetGroupName, NntpListGroups, NntpListNewGroups, NntpSelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetLastError Function

```
DWORD WINAPI NntpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **NntpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **NntpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **NntpSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those functions which call **NntpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetErrorString, NntpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetMessageIdArticle Function

```
LONG WINAPI NntpGetMessageIdArticle(
    HCLIENT hClient,
    LPCTSTR lpszMessageId
);
```

The **NntpGetMessageIdArticle** function returns the article number associated with a message identifier in the current newsgroup.

Parameters

hClient

Handle to the client session.

lpszMessageId

A pointer to a message identifier string.

Return Value

If the function succeeds, the return value is the article number. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

If the server uses 64-bit article IDs, your application should use the **NntpGetMessageIdArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetMessageIdArticleEx, NntpGetArticleMessageId, NntpListArticles

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetMessageIdArticleEx Function

```
BOOL WINAPI NntpGetMessageIdArticleEx(
    HCLIENT hClient,
    LPCTSTR lpszMessageId
    ULONGLONG *lpnArticleId
);
```

The **NntpGetMessageIdArticleEx** function returns the article number associated with a message identifier in the current newsgroup.

Parameters

hClient

Handle to the client session.

lpszMessageId

A pointer to a message identifier string.

lpnArticleId

A pointer to an unsigned 64-bit integer which will contain the article ID. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetMessageIdArticleEx** function returns a BOOL (a signed 32-bit integer) to indicate success or failure, and the article count is returned in a variable that is passed by reference to the function. If you are updating your code to use this function, make sure you also change how you check the return value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticleMessageIdEx, NntpListArticlesEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetMultiLine Function

```
INT WINAPI NntpGetMultiLine(
   HCLIENT hClient,
   LPBOOL lpbMultiLine
);
```

The **NntpGetMultiLine** function returns the value of the client multi-line flag in the specified boolean parameter.

Parameters

hClient

Handle to the client session.

lpbMultiLine

A pointer to a boolean variable. This variable will be set to the current value of the client's internal multi-line flag.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCommand, NntpGetResultCode, NntpGetResultString, NntpSetMultiLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetNextArticle Function

```
BOOL WINAPI NntpGetNextArticle(
   HCLIENT hClient,
   LPNEWSARTICLE lpArticle
);
```

The **NntpGetNextArticle** function returns information about the next article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticle

A pointer to a NEWSARTICLE structure which will contain information about the next available article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no more articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetNextArticle** function returns information about the next available article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetFirstArticle** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticle** function cannot be called while inside a loop calling **NntpGetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NttpGetArticle** for each article number to retrieve the article text.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstArticleEx** and **NntpGetNextArticleEx** functions.

Example

```
NEWSARTICLE newsArticle;
BOOL bResult;
INT nResult;

// List all articles in the current group
nResult = NntpListArticles(hClient, -1L, -1L);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
```

```
}

// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetNextArticle(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticle(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticle, NntpGetFirstArticle, NntpGetFirstArticleEx, NntpGetFirstGroup, NntpGetNextArticleEx, NntpGetNextGroup, NntpListArticles, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetNextArticleEx Function

```
BOOL WINAPI NntpGetNextArticleEx(
   HCLIENT hClient,
   LPNEWSARTICLEEX lpArticleEx
);
```

The **NntpGetNextArticleEx** function returns information about the next article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticle

A pointer to a NEWSARTICLEEX structure which will contain information about the next available article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no more articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetNextArticleEx** function returns information about the next available article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetFirstArticleEx** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access. This function provides support for servers which use 64-bit article IDs.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticleEx** function cannot be called while inside a loop calling **NntpGetNextArticleEx**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NttpGetArticleEx** for each article number to retrieve the article text.

Example

```
NEWSARTICLEEX newsArticle;
BOOL bResult;
INT nResult;

// List all articles in the current group
nResult = NntpListArticlesEx(hClient, (ULONGLONG)-1, (ULONGLONG)-1);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}
```

```
// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetNextArticleEx(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%I64u %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticleEx(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticleEx, NntpGetFirstArticleEx, NntpGetFirstGroupEx, NntpGetNextGroupEx, NntpListArticlesEx, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetNextGroup Function

```
BOOL WINAPI NntpGetNextGroup(
   HCLIENT hClient,
   LPNEWSGROUP lpGroup
);
```

The **NntpGetNextGroup** function returns information about the next available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroup

A pointer to a NEWSGROUP structure which will contain information about the next available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no more newsgroups available, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetNextGroup** function returns information about the next newsgroup on the server. This function is used in conjunction with the **NntpGetFirstGroup** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstGroupEx** and **NntpGetNextGroupEx** functions.

Example

```
NEWSGROUP newsGroup;
BOOL bResult;
INT nResult;

// List all available newsgroups
nResult = NntpListGroups(hClient);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}

// Get each newsgroup, printing the article range and
// the name of the group
```

```
bResult = NntpGetFirstGroup(hClient, &newsGroup);

while (bResult)
{
    _tprintf(_T("%ld %ld %s\n"), newsGroup.nFirstArticle,
                                 newsGroup.nLastArticle,
                                 newsGroup.szName);

    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstArticle, NntpGetFirstGroup, NntpGetFirstGroupEx, NntpGetNextArticle, NntpGetNextGroupEx, NntpListArticles, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetNextGroupEx Function

```
BOOL WINAPI NntpGetNextGroupEx(
   HCLIENT hClient,
   LPNEWSGROUPEX lpGroupEx
);
```

The **NntpGetNextGroupEx** function returns information about the next available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroup

A pointer to a NEWSGROUPEX structure which will contain information about the next available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no more newsgroups available, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetNextGroupEx** function returns information about the next newsgroup on the server. This function is used in conjunction with the **NntpGetFirstGroupEx** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. This function provides support for servers which use 64-bit article IDs.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
NEWSGROUPEX newsGroup;
BOOL bResult;
INT nResult;

// List all available newsgroups
nResult = NntpListGroups(hClient);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}

// Get each newsgroup, printing the article range and
// the name of the group
bResult = NntpGetFirstGroupEx(hClient, &newsGroup);
```

```
while (bResult)
{
    _tprintf(_T("%I64u %I64u %s\n"), newsGroup.nFirstArticle,
                                     newsGroup.nLastArticle,
                                     newsGroup.szName);

    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstArticle, NntpGetFirstGroup, NntpGetNextArticle, NntpListArticles, NntpListGroups, NntpListNewGroups, NntpSelectGroup, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetResultCode Function

```
INT WINAPI NntpGetResultCode(
    HCLIENT hClient
);
```

The **NntpGetResultCode** function reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCommand, NntpGetMultiLine, NntpGetResultString, NntpSetMultiLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetResultString Function

```
INT WINAPI NntpGetResultString(
    HCLIENT hClient,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

The **NntpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

The buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **NntpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpCommand, NntpGetMultiLine, NntpGetResultCode, NntpSetMultiLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetSecurityInformation Function

```
BOOL WINAPI NntpGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **NntpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (NntpGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
```

```
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpConnect, NntpDisconnect, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetStatus Function

```
INT WINAPI NntpGetStatus(
   HCLIENT hClient
);
```

The **NntpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
NNTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
NNTP_STATUS_CONNECT	The client is establishing a connection with the server.
NNTP_STATUS_READ	The client is reading data from the server.
NNTP_STATUS_WRITE	The client is writing data to the server.
NNTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpIsBlocking, NntpIsConnected, NntpIsReadable, NntpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetTimeout Function

```
INT WINAPI NntpGetTimeout(
   HCLIENT hClient
);
```

The **NntpGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetTransferStatus Function

```
INT WINAPI NntpGetTransferStatus(
   HCLIENT hClient,
   LPNNTPTRANSFERSTATUS lpStatus
);
```

The **NntpGetTransferStatus** function returns information about the current news article transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an NNTPTRANSFERSTATUS structure which contains information about the status of the current article transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetTransferStatus** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If no article is currently being retrieved or submitted to the server, this function will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the server uses 64-bit article IDs, your application should use the **NntpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpEnableEvents, NntpGetTransferStatusEx, NntpRegisterEvent, NNTPTRANSFERSTATUS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpGetTransferStatusEx Function

```
INT WINAPI NntpGetTransferStatusEx(
   HCLIENT hClient,
   LPNNTPTRANSFERSTATUSEX lpStatusEx
);
```

The **NntpGetTransferStatusEx** function returns information about the current news article transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an NNTPTRANSFERSTATUSEX structure which contains information about the status of the current article transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetTransferStatusEx** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. This function provides support for servers which use 64-bit article IDs. If no article is currently being retrieved or submitted to the server, this function will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpEnableEvents, NntpRegisterEvent, NNTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpInitialize Function

```
BOOL WINAPI NntpInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **NntpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **NntpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **NntpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **NntpInitialize** or **NntpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpConnect, NntpDisconnect, NntpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpIsBlocking Function

```
BOOL WINAPI NntpIsBlocking(
   HCLIENT hClient
);
```

The **NntpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **NntpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCancel, NntpCommand, NntpIsConnected, NntpIsReadable, NntpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpIsConnected Function

```
BOOL WINAPI NntpIsConnected(
    HCLIENT hClient
);
```

The **NntpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpIsBlocking, NntpIsReadable, NntpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpIsReadable Function

```
BOOL WINAPI NntpIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **NntpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetStatus, NntpIsBlocking, NntpIsConnected, NntpIsWritable, NntpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpIsWritable Function

```
BOOL WINAPI NntpIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **NntpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetStatus, NntpIsBlocking, NntpIsConnected, NntpIsReadable, NntpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpListArticles Function

```
INT WINAPI NntpListArticles(
   HCLIENT hClient,
   LONG nFirstArticle,
   LONG nLastArticle
);
```

The **NntpListArticles** function returns a list of articles in the currently selected newsgroup, within the specified article range.

Parameters

hClient

Handle to the client session.

nFirstArticle

The first newsgroup article to be returned in the list. If this value is -1, the list will begin with the first available article in the newsgroup.

nLastArticle

The last newsgroup article to be returned in the list. If the value is -1, the list will end with the last available article in the newsgroup.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. Use the **NntpGetFirstArticle** and **NntpGetNextArticle** functions to read the list of articles returned by the server.

If the server uses 64-bit article IDs, your application should use the **NntpListArticlesEx** function.

Example

```
NEWSARTICLE newsArticle;
BOOL bResult;
INT nResult;

// List all articles in the current group
nResult = NntpListArticles(hClient, -1L, -1L);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();

    NntpGetErrorString(dwError, szError, ASTRING);
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
    return;
}

// Get each article in the current newsgroup, printing the article
// number and the subject of the article
```

```
bResult = NntpGetNextArticle(hClient, &newsArticle);
while (bResult)
{
    _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticle(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetArticleRange, NntpGetCurrentArticle, NntpGetFirstArticle, NntpGetNextArticle, NntpListGroups, NntpListArticlesEx, NntpListNewGroups

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpListGroups Function

```
INT WINAPI NntpListGroups(
   HCLIENT hClient
);
```

The **NntpListGroups** function instructs the server to begin sending a list of newsgroups back to the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpListGroups** function is used in conjunction with the **NntpGetFirstGroup** and **NntpGetNextGroup** functions to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. To list only those newsgroups which have been added since a certain date, use the **NntpListNewGroups** function.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
NEWSGROUP newsGroup;
BOOL bResult;
INT nResult;

// List all available newsgroups
nResult = NntpListGroups(hClient);

if (nResult == NNTP_ERROR)
{
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();
    NntpGetErrorString(dwError, szError, ASTRING);
    fprintf(stderr, "Error %08x: %s\n", dwError, szError);
    return;
}

// Get each newsgroup, printing the article range and
// the name of the group
bResult = NntpGetFirstGroup(hClient, &newsGroup);
while (bResult)
{
    printf("%ld %ld %s\n", newsGroup.nFirstArticle,
                           newsGroup.nLastArticle,
                           newsGroup.szName);
    bResult = NntpGetNextGroup(hClient, &newsGroup);
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetArticleRange, NntpGetFirstGroup, NntpGetNextGroup, NntpListNewGroups, NntpSelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpListNewGroups Function

```
INT WINAPI NntpListNewGroups(
   HCLIENT hClient,
   LPCTSTR lpszLastUpdated,
   BOOL bLocalTime
);
```

The **NntpListNewGroups** function instructs the server to begin sending a list of newsgroups that were created since the specified date.

Parameters

hClient

Handle to the client session.

lpszLastUpdated

Pointer to a string which specifies the date and time that the list of newsgroups were last retrieved from the server. This parameter may be NULL or an empty string, in which case all available newsgroups will be listed by the server.

bLocalTime

A boolean value which indicates if the time specified in the ***lpszLastUpdated*** parameter is for the current timezone. If the value is non-zero, the time is assumed to be in the local timezone. If the value is zero, the time is assumed to be in Coordinated Universal Time (UTC).

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpListNewGroups** function is used in conjunction with the **NntpGetFirstGroup** and **NntpGetNextGroup** functions to enumerate all of the newsgroups that were added to the server since a specific date and time. Typically this is used to provide the user with a list of updated newsgroups to select. To list all of the newsgroups available on the server, use the **NntpListGroups** function.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
NEWSGROUP newsGroup;
LPCTSTR lpszUpdated = _T("1/1/2004 12:00 AM");
BOOL bResult;
INT nResult;

// List all newsgroups that were added after a
// specific date
nResult = NntpListNewGroups(hClient, lpszUpdated, TRUE);

if (nResult == NNTP_ERROR)
{
```

```
    TCHAR szError[ASTRING];
    DWORD dwError = NntpGetLastError();
    NntpGetErrorString(dwError, szError, ASTRING);
    fprintf(stderr, "Error %08x: %s\n", dwError, szError);
    return;
}

// Get each newsgroup, printing the article range and
// the name of the group
bResult = NntpGetFirstGroup(hClient, &newsGroup);
while (bResult)
{
    printf("%ld %ld %s\n", newsGroup.nFirstArticle,
                           newsGroup.nLastArticle,
                           newsGroup.szName);
    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticleRange, NntpGetFirstGroup, NntpGetNextGroup, NntpListGroups, NntpSelectGroup

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpOpenArticle Function

```
INT WINAPI NntpOpenArticle(
   HCLIENT hClient,
   LONG nArticleId
);
```

The **NntpOpenArticle** function opens the specified article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

An integer value that specifies which article in the current newsgroup to retrieve. This value may be zero, which specifies that the current article should be opened.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpOpenArticle** function sends a request to begin returning the contents of the specified article. The **NntpRead** function is used to read the article contents after it has been opened. When the complete article has been read, the **NntpCloseArticle** function must be called to close the article and complete the request.

It is recommended most applications use the **NntpGetArticle** function, which will retrieve the complete article in a single function call. This function is typically only used if the application needs to modify the contents of the article as it is being read. If you wish to download an article and store it in a file on the local system, use the **NntpStoreArticle** function.

If the server uses 64-bit article IDs, your application should use the **NntpOpenArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCloseArticle, NntpGetArticle, NntpGetArticleRange, NntpGetCurrentArticle, NntpListArticles, NntpOpenArticleByMessageId, NntpOpenNextArticle, NntpOpenPreviousArticle, NntpRead, NntpStoreArticle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpOpenArticleEx Function

```
INT WINAPI NntpOpenArticleEx(
   HCLIENT hClient,
   ULONGLONG nArticleId
);
```

The **NntpOpenArticle** function opens the specified article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

An unsigned 64-bit integer that specifies which article in the current newsgroup to retrieve. This value may be zero, which specifies that the current article should be opened.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpOpenArticleEx** function sends a request to begin returning the contents of the specified article. This function provides support for servers which use 64-bit article IDs. The **NntpRead** function is used to read the article contents after it has been opened. When the complete article has been read, the **NntpCloseArticle** function must be called to close the article and complete the request.

It is recommended most applications use the **NntpGetArticleEx** function, which will retrieve the complete article in a single function call. This function is typically only used if the application needs to modify the contents of the article as it is being read. If you wish to download an article and store it in a file on the local system, use the **NntpStoreArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCloseArticle, NntpGetArticleEx, NntpGetArticleRangeEx, NntpGetCurrentArticleEx, NntpListArticlesEx, NntpOpenArticleByMessageId, NntpOpenNextArticle, NntpOpenPreviousArticle, NntpRead, NntpStoreArticleEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpOpenArticleByMessageId Function

```
INT WINAPI NntpOpenArticleByMessageId(
    HCLIENT hClient,
    LPCTSTR lpszMessageId
);
```

The **NntpOpenArticleByMessageId** function opens the article specified by the message identifier string.

Parameters

hClient

Handle to the client session.

lpszMessageId

Pointer to a string which contains the message identifier for the article in the current newsgroup.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpCloseArticle, NntpGetArticle, NntpGetArticleRange, NntpGetCurrentArticle, NntpListArticles, NntpOpenArticle, NntpOpenNextArticle, NntpOpenPreviousArticle, NntpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpOpenNextArticle Function

```
INT WINAPI NntpOpenNextArticle(
    HCLIENT hClient
);
```

The **NntpOpenNextArticle** function opens the next available article in the current newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCloseArticle, NntpGetArticle, NntpGetArticleRange, NntpGetCurrentArticle, NntpListArticles, NntpOpenArticleByMessageId, NntpOpenPreviousArticle, NntpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpOpenPreviousArticle Function

```
INT WINAPI NntpOpenPreviousArticle(
    HCLIENT hClient
);
```

The **NntpOpenPreviousArticle** function opens the previous article in the current newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCloseArticle, NntpGetArticle, NntpGetArticleRange, NntpGetCurrentArticle, NntpListArticles, NntpOpenArticleByMessageId, NntpOpenNextArticle, NntpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpPostArticle Function

```
INT WINAPI NntpPostArticle(
   HCLIENT hClient,
   LPCTSTR lpBuffer,
   DWORD dwLength,
   DWORD dwReserved
);
```

The **NntpPostArticle** function post the contents of the specified buffer to the server as a new article in the current newsgroup.

Parameters

hClient

Handle to the client session.

lpBuffer

A pointer to a character buffer which contains the article to be posted to the currently selected newsgroup.

dwLength

Specifies the length of the string which contains the article. If this parameter is -1, the actual length of the string is calculated by searching the buffer for a terminating null byte.

dwReserved

Reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpPostArticle** function is used to post the contents of the specified buffer to the server as a new article in the current newsgroup. Not all newsgroups permit new articles to be posted, and some newsgroups may require that you email the article to a moderator for approval instead of posting directly to the group. It may be required that the client authenticate itself using the **NntpAuthenticate** function prior to posting the article.

A news article is similar to an email message in that it contains one or more header fields, followed by an empty line, followed by the body of the article. Each line of text should be terminated by a carriage return/linefeed sequence of characters. The Mail Message library can be used to compose a message if needed. Note that the article header must contain a header field named "Newsgroups" with a value that specifies the newsgroup or newsgroups the article is being posted to. If this header field is missing, the news server will reject the article.

This function will cause the current thread to block until the transfer has completed, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **NntpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetTransferStatus, NntpIsBlocking, NntpIsWritable, NntpRead, NntpReset, NntpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpRead Function

```
INT WINAPI NntpRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **NntpRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When **NntpRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCloseArticle, NntpIsBlocking, NntpIsConnected, NntpIsReadable, NntpOpenArticle, NntpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpRegisterEvent Function

```
INT WINAPI NntpRegisterEvent(
  HCLIENT hClient,
  UINT nEvent,
  NNTPEVENTPROC lpEventProc,
  DWORD_PTR dwParam
);
```

The **NntpRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
NNTP_EVENT_CONNECT	The connection to the server has completed.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_LISTARTICLE	The client is retrieving a list of articles from the server and information about another article has been processed. The

	client should use this event to retrieve the next available article from the server.
NNTP_EVENT_LASTARTICLE	The client is retrieving a list of articles from the server and all of the available articles have been processed. This event is generated when there are no further articles.
NNTP_EVENT_LISTGROUP	The client is retrieving a list of newsgroups from the server and information about another group has been processed. The client should use this event to retrieve the next available newsgroup from the server.
NNTP_EVENT_LASTGROUP	The client is retrieving a list of newsgroups from the server and all of the available groups have been processed. This event occurs when there are no further newsgroups.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **NntpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpRegisterEvent** function associates a callback function with a specific event. The event handler is an **NntpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a news article is being uploaded or downloaded. The NNTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one NNTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpDisableEvents, NntpEnableEvents, NntpEventProc, NntpFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpReset Function

```
INT WINAPI NntpReset(
   HCLIENT hClient
);
```

The **NntpReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The client cannot be reset while it is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCancel, NntpIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpSelectGroup Function

```
INT WINAPI NntpSelectGroup(
   HCLIENT hClient,
   LPCTSTR lpszGroupName
);
```

The **NntpSelectGroup** function selects the specified newsgroup from which articles will be retrieved.

Parameters

hClient

Handle to the client session.

lpszGroupName

Pointer to a string which specifies the newsgroup to be selected. This value may be NULL, in which case the current newsgroup is unchanged, but the article count is updated.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function selects the newsgroup and obtains a description and the first and last article numbers for that group.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetGroupName, NntpGetGroupTitle, NntpListGroups, NntpListNewGroups

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpSetLastError Function

```
VOID WINAPI NntpSetLastError(
    DWORD dwErrorCode
);
```

The **NntpSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those functions which call **NntpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **NntpGetLastError** function. The use of **NntpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpGetErrorString, NntpGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpSetMultiLine Function

```
INT WINAPI NntpSetMultiLine(
   HCLIENT hClient,
   BOOL bMultiLine
);
```

The **NntpSetMultiLine** function sets the client multi-line flag into the specified value.

Parameters

hClient

Handle to the client session.

bMultiLine

A boolean flag which determines if the client is processing multiple lines of data as the result of a command.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

The **NntpSetMultiLine** function should only be used in conjunction with the **NntpCommand** function. If a command is issued which would result in multiple lines of output, the multi-line flag must be set TRUE. The multi-line flag must be set after each command, since it is reset to FALSE with each command that is sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCommand, NntpGetMultiLine, NntpGetResultCode, NntpGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpSetTimeout Function

```
INT WINAPI NntpSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **NntpSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpConnect, NntpGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpStoreArticle Function

```
INT WINAPI NntpStoreArticle(
   HCLIENT hClient,
   LONG nArticleId,
   LPCTSTR lpszFileName
);
```

The **NntpStoreMessage** function retrieves an article from the current newsgroup and stores it in a local file.

Parameters

hClient

Handle to the client session.

nArticleId

An integer value that specifies the article to be downloaded.

lpszFileName

Pointer to a string which specifies the file that the article will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the article.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpStoreArticle** function provides a method of retrieving and storing an article on the local system. The contents of the article is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode. If you wish to download the article to memory rather than a file, use the **NntpGetArticle** function.

If event handling is enabled, the NNTP_EVENT_PROGRESS event will fire periodically during the transfer of the article to the local system. An application can determine how much of the article has been retrieved by calling the **NntpGetTransferStatus** function.

If the server uses 64-bit article IDs, your application should use the **NntpStoreArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticle, NntpGetArticleHeaders, NntpGetTransferStatus, NntpStoreArticleEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpStoreArticleEx Function

```
INT WINAPI NntpStoreArticleEx(
   HCLIENT hClient,
   ULONGLONG nArticleId,
   LPCTSTR lpszFileName,
   DWORD dwReserved
);
```

The **NntpStoreMessageEx** function retrieves an article from the current newsgroup and stores it in a local file.

Parameters

hClient

Handle to the client session.

nArticleId

An unsigned 64-integer value which specifies the article to be downloaded.

lpszFileName

Pointer to a string which specifies the file that the article will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the article.

dwReserved

Number of the article to retrieve. This value must be greater than zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpStoreArticleEx** function provides a method of retrieving and storing an article on the local system. This function provides support for servers which use 64-bit article IDs.

The contents of the article is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode. If you wish to download the article to memory rather than a file, use the **NntpGetArticle** function.

If event handling is enabled, the NNTP_EVENT_PROGRESS event will fire periodically during the transfer of the article to the local system. An application can determine how much of the article has been retrieved by calling the **NntpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetArticleEx, NntpGetArticleHeadersEx, NntpGetTransferStatusEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpUninitialize Function

```
VOID WINAPI NntpUninitialize();
```

The **NntpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **NntpInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **NntpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **NntpUninitialize** for every successful call to **NntpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpDisconnect, NntpInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NntpWrite Function

```
INT WINAPI NntpWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **NntpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the NNTP_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

See Also

NntpCreateArticle, NntpIsBlocking, NntpRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network News Transfer Protocol Data Structures

- NEWSARTICLE
- NEWSARTICLEX
- NEWSGROUP
- NEWSGROUPEX
- NNTPTRANSFERSTATUS
- NNTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSARTICLE

This structure is used by the **NntpGetFirstArticle** and **NntpGetNextArticle** functions to return information about news articles.

```
typedef struct _NEWSARTICLE
{
    LONG        nArticleId;
    LONG        nBytes;
    LONG        nLines;
    TCHAR       szSubject[NNTP_MAXSUBJLEN];
    TCHAR       szAuthor[NNTP_MAXAUTHLEN];
    TCHAR       szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR       szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME  stPosted;
} NEWSARTICLE, *LPNEWSARTICLE;
```

Members

nArticleId

A long integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A null-terminated string which specifies the subject of the article.

szAuthor

A null-terminated string which specifies the email address of the user who posted the article.

szMessageId

A null-terminated string which specifies the message ID for the article.

szReferences

A null-terminated string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Remarks

If the server uses 64-bit article IDs your application should use the NEWSARTICLEEX structure and related functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstArticle, NntpGetNextArticle, NntpListArticles, NEWSARTICLE, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSARTICLEEX

This structure is used by the **NntpGetFirstArticleEx** and **NntpGetNextArticleEx** functions to return information about news articles.

```
typedef struct _NEWSARTICLEEX
{
    ULONGLONG   nArticleId;
    ULONG       nBytes;
    ULONG       nLines;
    TCHAR       szSubject[NNTP_MAXSUBJLEN];
    TCHAR       szAuthor[NNTP_MAXAUTHLEN];
    TCHAR       szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR       szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME  stPosted;
} NEWSARTICLEEX, *LPNEWSARTICLEEX;
```

Members

nArticleId

An unsigned 64-bit integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A null-terminated string which specifies the subject of the article.

szAuthor

A null-terminated string which specifies the email address of the user who posted the article.

szMessageId

A null-terminated string which specifies the message ID for the article.

szReferences

A null-terminated string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstArticleEx, NntpGetNextArticleEx, NntpListArticlesEx, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSGROUP

This structure is used by the **NntpGetFirstGroup** and **NntpGetNextGroup** functions to return information about the available newsgroups.

```
typedef struct _NEWSGROUP
{
    LONG        nFirstArticle;
    LONG        nLastArticle;
    DWORD       dwAccess;
    TCHAR       szName[NNTP_MAXGRPNAMLEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

A long integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

A long integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Value	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A null-terminated string which specifies the name of the newsgroup.

Remarks

If the server uses 64-bit article IDs your application should use the NEWSGROUPEX structure and related functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstGroup, NntpGetNextGroup, NEWSARTICLE, NEWSARTICLEEX, NEWSGROUPEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NEWSGROUPEX

This structure is used by the **NntpGetFirstGroupEx** and **NntpGetNextGroupEx** functions to return information about the available newsgroups.

```
typedef struct _NEWSGROUPEX
{
    ULONGLONG   nFirstArticle;
    ULONGLONG   nLastArticle;
    DWORD       dwAccess;
    TCHAR       szName[NNTP_MAXGRPNAMLEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

An unsigned 64-bit integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

An unsigned 64-bit integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Value	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A null-terminated string which specifies the name of the newsgroup.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnwsv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NntpGetFirstGroupEx, NntpGetNextGroupEx, NEWSARTICLE, NEWSARTICLEEX, NEWSGROUP

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NNTPTRANSFERSTATUS

This structure is used by the **NntpGetTransferStatus** function to return information about an article transfer in progress. If the server uses 64-bit article IDs your application should use the NNTPTRANSFERSTATUSEX structure and related functions.

```
typedef struct _NNTPTRANSFERSTATUS
{
   LONG    nArticleId;
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} NNTPTRANSFERSTATUS, *LPNNTPTRANSFERSTATUS;
```

Members

nArticleId

A signed integer that specifies the article ID of the current article that is being transferred. If an article is being posted, this value will be zero.

dwBytesTotal

An unsigned integer that specifies the total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

An unsigned integer that specifies the total number of bytes that have been copied.

dwBytesPerSecond

An unsigned integer that specifies the average number of bytes that have been copied per second.

dwTimeElapsed

An unsigned integer that specifies the number of seconds that have elapsed since the transfer started.

dwTimeEstimated

An unsigned integer that specifies the estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

NntpEnableEvents, NntpGetTransferStatus, NntpRegisterEvent, NNTPTRANSFERSTATUS, NNTPTRANSFERSTATUSEX

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NNTPTRANSFERSTATUSEX

This structure is used by the **NntpGetTransferStatusEx** function to return information about an article transfer in progress. This structure is intended for use with servers that return 64-bit article IDs.

```
typedef struct _NNTPTRANSFERSTATUSEX
{
   ULONGLONG nArticleId;
   DWORD     dwBytesTotal;
   DWORD     dwBytesCopied;
   DWORD     dwBytesPerSecond;
   DWORD     dwTimeElapsed;
   DWORD     dwTimeEstimated;
} NNTPTRANSFERSTATUSEX, *LPNNTPTRANSFERSTATUSEX;
```

Members

nArticleId

An unsigned 64-bit integer that specifies the article ID of the current article that is being transferred. If an article is being posted, this value will be zero.

dwBytesTotal

An unsigned integer that specifies the total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

An unsigned integer that specifies the total number of bytes that have been copied.

dwBytesPerSecond

An unsigned integer that specifies the average number of bytes that have been copied per second.

dwTimeElapsed

An unsigned integer that specifies the number of seconds that have elapsed since the transfer started.

dwTimeEstimated

An unsigned integer that specifies the estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

NntpEnableEvents, NntpGetTransferStatusEx, NntpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol Library

Query a network time server for the current date and time.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSTIMV12.DLL
Version	12.0.3024.1286
LibID	79C32362-35E2-4115-9F08-043C7F982BB8
Import Library	CSTIMV12.LIB
Dependencies	None
Standards	RFC 868, RFC 5905

Overview

The Network Time Protocol library provides an interface for obtaining the current date and time from a server. The time is returned in Coordinated Universal Time (UCT) and can be adjusted for the local host's timezone. The time value is returned as the number of seconds which have elapsed since midnight, January 1, 1900. The library supports the Network Time Protocol (NTP) as specified in RFC 5905 and the TIME protocol described in RFC 868.

It is recommended you update your applications to call the functions which use 64-bit integer time values and use the NTP v4 protocol, which is supported in the current version of this API. To use NTP v4, provide the name of a valid NTP server and specify port 123. Although the API continues to support the TIME protocol on port 37, its use has been deprecated and applications should be updated to use NTP which is considered a more reliable method for obtaining the current time over the Internet.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy

development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol Functions

Function	Description
ConvertNetworkTime	Convert 32-bit network time to system time
ConvertNetworkTime64	Convert a 64-bit network time to system time
ConvertSystemTime	Convert system time to a 32-bit network time
ConvertSystemTime64	Convert system time to a 64-bit network time
FormatNetworkTime	Return the specified 32-bit network time as a string formatted for the current locale.
FormatNetworkTime64	Return the specified 64-bit network time as a string formatted for the current locale.
GetNetworkTime	Return the current 32-bit network time from the server
GetNetworkTime64	Return the current 64-bit network time from the server
QueryNetworkTime	Query the server for the network time
ReadNetworkTime	Read the network time returned by the server
ReadNetworkTime64	Read the 64-bit network time returned by the server
TimeDisableTrace	Disable logging of network function calls
TimeEnableTrace	Enable logging of network function calls to a file
TimeGetErrorString	Return a description for the specified error code
TimeGetLastError	Return the last error code
TimeInitialize	Initialize the library and validates the specified license key at runtime
TimeSetLastError	Set the last error code
TimeUninitialize	Terminate use of the library by the application
UpdateLocalTime	Update the local system time with a 32-bit network time
UpdateLocalTime64	Update the local system time with a 64-bit network time

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ConvertNetworkTime Function

```
BOOL WINAPI ConvertNetworkTime(
   DWORD dwNetworkTime,
   LPSYSTEMTIME lpSystemTime,
   BOOL bLocalTime
);
```

The **ConvertNetworkTime** function converts the specified network time, adjusting for the local timezone if required. The network time is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

dwNetworkTime

An unsigned 32-bit integer value which specifies the network time. This value should specify the number of seconds since midnight, January 1, 1900 UTC.

lpSystemTime

A pointer to a SYSTEMTIME structure which will be modified for the specified network time.

bLocalTime

A boolean flag that is used to specify if the network time should be adjusted for the local timezone.

Return Value

If the network time could be converted, the function returns a non-zero value. If the network time cannot be converted, or the pointer to the SYSTEMTIME structure is invalid, the function will return zero.

Remarks

This function is included for backwards compatibility and should not be used with new applications. The 32-bit network time value can only represent a date and time up to the year 2036. Use the **ConvertNetworkTime64** function which accepts a 64-bit integer and avoids potential problems with the 32-bit value rolling over.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime64, ConvertSystemTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ConvertNetworkTime64 Function

```
BOOL WINAPI ConvertNetworkTime64(
   ULONGLONG ulNetworkTime,
   LPSYSTEMTIME lpSystemTime,
   BOOL bLocalTime
);
```

The **ConvertNetworkTime64** function converts the specified network time, adjusting for the local timezone if required. The network time is a 64-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

ulNetworkTime

An unsigned 64-bit integer value which specifies the network time. This value should specify the number of seconds since midnight, January 1, 1900 UTC.

lpSystemTime

A pointer to a SYSTEMTIME structure which will be modified for the specified network time.

bLocalTime

A boolean flag that is used to specify if the network time should be adjusted for the local timezone.

Return Value

If the network time could be converted, the function returns a non-zero value. If the network time cannot be converted, or the pointer to the SYSTEMTIME structure is invalid, the function will return zero.

Remarks

This version of the function supports 64-bit network time values which can represent a date and time beyond the year 2036.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime, ConvertSystemTime64

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ConvertSystemTime Function

```
DWORD WINAPI ConvertSystemTime(
   LPSYSTEMTIME lpSystemTime
);
```

The **ConvertSystemTime** function converts the specified system time to network time. The network time value is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

lpSystemTime

A pointer to a SYSTEMTIME structure which will be modified for the specified network time.

Return Value

If the system time could be converted, the function returns the number of seconds since midnight, 1 January 1900 UTC. If the pointer to the SYSTEMTIME structure is invalid, or the structure contains invalid data, the function will return zero.

Remarks

This function is included for backwards compatibility and should not be used with new applications. The 32-bit network time value can only represent a date and time up to the year 2036. Use the **ConvertSystemTime64** function which returns a 64-bit integer and avoids potential problems with the 32-bit value rolling over.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime, ConvertNetworkTime64, UpdateLocalTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ConvertSystemTime64 Function

```
ULONGLONG WINAPI ConvertSystemTime64(
   LPSYSTEMTIME lpSystemTime
);
```

The **ConvertSystemTime64** function converts the specified system time to network time. The network time value is a 64-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

lpSystemTime

A pointer to a SYSTEMTIME structure which will be modified for the specified network time.

Return Value

If the system time could be converted, the function returns the number of seconds since midnight, 1 January 1900 UTC. If the pointer to the SYSTEMTIME structure is invalid, or the structure contains invalid data, the function will return zero.

Remarks

This version of the function supports 64-bit network time values which can represent a date and time beyond the year 2036.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime64, ConvertSystemTime, UpdateLocalTime64

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FormatNetworkTime Function

```
INT WINAPI FormatNetworkTime(
   DWORD dwNetworkTime,
   LPTSTR lpszDateTime,
   INT nMaxLength,
   BOOL bLocalTime
);
```

The **FormatNetworkTime** function converts a 32-bit network time value to a string which contains the date and time formatted for the current locale.

Parameters

dwNetworkTime

An unsigned 32-bit integer value which specifies the network time. This value should specify the number of seconds since midnight, January 1, 1900 UTC.

lpszDateTime

A null-terminated string buffer which will contain the formatted date and time. This buffer must be large enough to store both date and time values, including the terminating null character. The date is formatted using the current system locale's short date format. The time is formatted to use a 24-hour clock and includes the seconds value.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. If the buffer is not large enough to contain the entire date and time string, the function will fail and return a value of zero.

bLocalTime

An integer boolean value which specifies if the network time should be adjusted for the local timezone. If this parameter is zero, the date and time are formatted using Coordinated Universal Time (UTC). If this parameter is non-zero, the network time value will be adjusted for the current timezone and will return a localized time value. The current timezone is determined by the configuration of the local system.

Return Value

If successful, the function returns the number of characters copied into the string buffer, not including the terminating null character. If the function fails, it returns a value of zero. Call the **TimeGetLastError** function for more information about the cause of the failure.

Remarks

The **FormatNetworkTime** function is a convenient way to convert a numeric network time value to a string value which can be displayed to the user. The actual format of the string is determined by the local system configuration and locale. For example, in North America the date string will be returned using a month/day/year format, while European locales may return the date in a day/month/year format.

This function is included for backwards compatibility and should not be used with new applications. The 32-bit network time value can only represent a date and time up to the year 2036. Use the **FormatNetworkTime64** function which accepts a 64-bit integer and avoids potential problems with the 32-bit value rolling over.

Example

```
// Request the time from a server and write it to standard output
DWORD dwNetworkTime = GetNetworkTime(_T("time.nist.gov"), TIME_PORT_DEFAULT,
TIME_TIMEOUT);

if (dwNetworkTime != 0)
{
    TCHAR szDateTime[64];
    INT nLength = FormatNetworkTime(dwNetworkTime, szDateTime, 64, TRUE);

    if (nLength > 0)
        _tprintf(_T("The local date and time is %s\n"), szDateTime);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertNetworkTime, FormatNetworkTime64, GetNetworkTime, UpdateLocalTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

FormatNetworkTime64 Function

```
INT WINAPI FormatNetworkTime64(
   ULONGLONG ulNetworkTime,
   LPTSTR lpszDateTime,
   INT nMaxLength,
   BOOL bLocalTime
);
```

The **FormatNetworkTime64** function converts a 64-bit network time value to a string which contains the date and time formatted for the current locale.

Parameters

ulNetworkTime

An unsigned 64-bit integer value which specifies the network time. This value should specify the number of seconds since midnight, January 1, 1900 UTC.

lpszDateTime

A null-terminated string buffer which will contain the formatted date and time. This buffer must be large enough to store both date and time values, including the terminating null character. The date is formatted using the current system locale's short date format. The time is formatted to use a 24-hour clock and includes the seconds value.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. If the buffer is not large enough to contain the entire date and time string, the function will fail and return a value of zero.

bLocalTime

An integer boolean value which specifies if the network time should be adjusted for the local timezone. If this parameter is zero, the date and time are formatted using Coordinated Universal Time (UTC). If this parameter is non-zero, the network time value will be adjusted for the current timezone and will return a localized time value. The current timezone is determined by the configuration of the local system.

Return Value

If successful, the function returns the number of characters copied into the string buffer, not including the terminating null character. If the function fails, it returns a value of zero. Call the **TimeGetLastError** function for more information about the cause of the failure.

Remarks

The **FormatNetworkTime64** function is a convenient way to convert a numeric network time value to a string value which can be displayed to the user. The actual format of the string is determined by the local system configuration and locale. For example, in North America the date string will be returned using a month/day/year format, while European locales may return the date in a day/month/year format.

Example

```
// Request the time from a server and write it to standard output
ULONGLONG ulNetworkTime = GetNetworkTime64(_T("pool.ntp.org"), TIME_PORT_DEFAULT,
TIME_TIMEOUT);
```

```
if (ulNetworkTime != 0)
{
    TCHAR szDateTime[64];
    INT nLength = FormatNetworkTime64(ulNetworkTime, szDateTime, 64, TRUE);

    if (nLength > 0)
        _tprintf(_T("The local date and time is %s\n"), szDateTime);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertNetworkTime64, FormatNetworkTime, GetNetworkTime64, UpdateLocalTime64

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetNetworkTime Function

```
DWORD WINAPI GetNetworkTime(
   LPCTSTR lpszHostName,
   UINT nHostPort,
   UINT nTimeout
);
```

The **GetNetworkTime** function returns the network time from the specified host. The network time is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

lpszHostName

The name of the server. The host must be running a time server that complies with the Network Time Protocol (NTP) specification outlined in RFC 5905.

nHostPort

The port the time server is running on. A value of zero indicates that the default port number for the NTP service should be used. If the application specifies port 37, the library will attempt to use the legacy TIME protocol defined in RFC 868. Support for this protocol is deprecated and is only provided for backwards compatibility.

nTimeout

The number of seconds that the function will wait for a response from the server.

Return Value

If the function succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the function was unable to obtain the time from the specified host, it returns zero.

Remarks

This function is included for backwards compatibility and should not be used with new applications. The 32-bit network time value can only represent a date and time up to the year 2036. Use the **GetNetworkTime64** function which returns a 64-bit integer and avoids potential problems with the 32-bit value rolling over.

This function will cause the calling thread to block until the time is returned by the server, or the operation times out. If you do not want the current thread to block, you can create a worker thread and call this function from within that thread. The blocking network call in the worker thread will not interefere with any other threads in your application.

In most cases, it is preferable to use the NTP pool servers which distribute requests over a number of different servers:

- **pool.ntp.org** is a global load-balanced pool of time servers
- **us.pool.ntp.org** servers are located in the United States
- **eu.pool.ntp.org** servers are located in Europe
- **asia.pool.ntp.org** servers are located in Asia

There are also load balanced pools of time servers available in specific countries prefixed by their country code. For example, the **de.pool.ntp.org** servers are located in Germany and the **uk.pool.ntp.org** servers are located in the United Kingdom. Regardless of their location, the time values they return are always in Coordinated Universal Time (UTC) and are not localized based on the

physical location of the server.

In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a-g.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b-g.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a-b.nist.gov	132.163.96.1	Boulder, Colorado
time-b-b.nist.gov	132.163.96.2	Boulder, Colorado
time-c-b.nist.gov	132.163.96.3	Boulder, Colorado

Time servers are also commonly maintained by government agencies and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they support the standard time service available on port 123.

If you are requesting the time from a public server, you should ensure that your application does not send too many requests over a short period of time. In most cases, you should not query a time server more than once per minute. Most public time servers will block the IP addresse of a client which sends too many requests.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch. NTP servers return the time as the number of seconds since January 1, 1900.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertNetworkTime, FormatNetworkTime, GetNetworkTime64, QueryNetworkTime, ReadNetworkTime, TimeInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

GetNetworkTime64 Function

```
ULONGLONG WINAPI GetNetworkTime64(
    LPCTSTR lpszHostName,
    UINT nHostPort,
    UINT nTimeout
);
```

The **GetNetworkTime64** function returns the network time from the specified host. The network time is a 64-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

lpszHostName

The name of the server. The host must be running a time server that complies with the Network Time Protocol (NTP) specification outlined in RFC 5905.

nHostPort

The port the time server is running on. A value of zero indicates that the default port number for the NTP service should be used. If the application specifies port 37, the library will attempt to use the legacy TIME protocol defined in RFC 868. Support for this protocol is deprecated and is only provided for backwards compatibility.

nTimeout

The number of seconds that the function will wait for a response from the server.

Return Value

If the function succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the function was unable to obtain the time from the specified host, it returns zero.

Remarks

The **GetNetworkTime64** function will cause the calling thread to block until the time is returned by the server, or the operation times out. If you do not want the current thread to block, you can create a worker thread and call this function from within that thread. The blocking network call in the worker thread will not interefere with any other threads in your application.

In most cases, it is preferable to use the NTP pool servers which distribute requests over a number of different servers:

- **pool.ntp.org** is a global load-balanced pool of time servers
- **us.pool.ntp.org** servers are located in the United States
- **eu.pool.ntp.org** servers are located in Europe
- **asia.pool.ntp.org** servers are located in Asia

There are also load balanced pools of time servers available in specific countries prefixed by their country code. For example, the **de.pool.ntp.org** servers are located in Germany and the **uk.pool.ntp.org** servers are located in the United Kingdom. Regardless of their location, the time values they return are always in Coordinated Universal Time (UTC) and are not localized based on the physical location of the server.

In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a-g.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b-g.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a-b.nist.gov	132.163.96.1	Boulder, Colorado
time-b-b.nist.gov	132.163.96.2	Boulder, Colorado
time-c-b.nist.gov	132.163.96.3	Boulder, Colorado

Time servers are also commonly maintained by government agencies and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they support the standard time service available on port 123.

If you are requesting the time from a public server, you should ensure that your application does not send too many requests over a short period of time. In most cases, you should not query a time server more than once per minute. Most public time servers will block the IP addresse of a client which sends too many requests.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch. NTP servers return the time as the number of seconds since January 1, 1900.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertNetworkTime64, FormatNetworkTime64, GetNetworkTime, QueryNetworkTime, ReadNetworkTime64, TimeInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

QueryNetworkTime Function

```
HCLIENT WINAPI QueryNetworkTime(
    LPCTSTR lpszHostName,
    UINT nHostPort,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **QueryNetworkTime** function connects to the specified server and issues a request for the current network time. The handle is passed as a parameter to the **ReadNetworkTime64** function to read the value returned by the server.

Parameters

lpszHostName

The name of the server. The host must be running a time server that complies with the Network Time Protocol (NTP) specification outlined in RFC 5905.

nHostPort

The port the time server is running on. A value of zero indicates that the default port number for the NTP service should be used. If the application specifies port 37, the library will attempt to use the legacy TIME protocol defined in RFC 868. Support for this protocol is deprecated and is only provided for backwards compatibility.

hEventWnd

A handle to the window that will receive the notification that the network time has been returned by the server.

uEventMsg

The notification message that will be sent to the window, indicating that the client has received the network time from the server.

Return Value

If the function succeeds, it returns a handle which can be used with the obtain the network time. If the function fails, it returns INVALID_CLIENT. To get extended error information, call **TimeGetLastError**.

Remarks

The **QueryNetworkTime** function creates an asynchronous datagram socket which sends a request to the time server for the current time. Use the **ReadNetworkTime64** function to read the response from the server and return the time as a 64-bit value which specifies the number of seconds elapsed since midnight, January 1, 1900. Typically this would be done in response to a TIME_EVENT_REPLY event message.

It is important to note the event notifications are sent as Windows messages which requires a window handle and the application must have a message pump. For console applications, you should use the **GetNetworkTime64** function instead. If you do not want to block the main thread while waiting for a response, you can create a background worker thread and send the request from within that thread. The blocking network call in the worker thread will not interefere with any other threads in your application.

In most cases, it is preferable to use the NTP pool servers which distribute requests over a number of different servers:

- **pool.ntp.org** is a global load-balanced pool of time servers
- **us.pool.ntp.org** servers are located in the United States
- **eu.pool.ntp.org** servers are located in Europe
- **asia.pool.ntp.org** servers are located in Asia

There are also load balanced pools of time servers available in specific countries prefixed by their country code. For example, the **de.pool.ntp.org** servers are located in Germany and the **uk.pool.ntp.org** servers are located in the United Kingdom. Regardless of their location, the time values they return are always in Coordinated Universal Time (UTC) and are not localized based on the physical location of the server.

In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a-g.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b-g.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a-b.nist.gov	132.163.96.1	Boulder, Colorado
time-b-b.nist.gov	132.163.96.2	Boulder, Colorado
time-c-b.nist.gov	132.163.96.3	Boulder, Colorado

Time servers are also commonly maintained by government agencies and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they support the standard time service available on port 123.

If you are requesting the time from a public server, you should ensure that your application does not send too many requests over a short period of time. In most cases, you should not query a time server more than once per minute. Most public time servers will block the IP addresse of a client which sends too many requests.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
TIME_EVENT_QUERY	The connection to the server has completed and the request for the time has been sent. The client should wait for the response from the server.
TIME_EVENT_REPLY	The server has replied to the request for the time. The client should call the ReadNetworkTime function to read the value returned by the server.
TIME_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TIME_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or

	refuse to accept further commands from the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

ConvertNetworkTime64, GetNetworkTime64, ReadNetworkTime64, UpdateLocalTime64

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ReadNetworkTime Function

```
DWORD WINAPI ReadNetworkTime(
    HCLIENT hClient
);
```

The **ReadNetworkTime** function returns a 32-bit network time value sent from the server.

Parameters

hClient

The client handle.

Return Value

If the function succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the function was unable to read the time from the specified host, it returns zero.

Remarks

The **ReadNetworkTime** function reads 32-bit the time value returned by the server using the handle returned by the **QueryNetworkTime** function. This function is included for backwards compatibility and should not be used with new applications. The 32-bit network time value can only represent a date and time up to the year 2036. Use the **ReadNetworkTime64** function which returns a 64-bit integer and avoids potential problems with the 32-bit value rolling over. To obtain the time from a server synchronously, use the **GetNetworkTime64** function instead.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch. NTP servers return the time as the number of seconds since January 1, 1900.

To convert the 32-bit time value to a SYSTEMTIME structure, use the **ConvertNetworkTime** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime, FormatNetworkTime, GetNetworkTime64, ReadNetworkTime64, QueryNetworkTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

ReadNetworkTime64 Function

```
ULONGLONG WINAPI ReadNetworkTime64(
    HCLIENT hClient
);
```

The **ReadNetworkTime64** function returns a 64-bit network time value sent from the server.

Parameters

hClient

The client handle.

Return Value

If the function succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the function was unable to read the time from the specified host, it returns zero.

Remarks

The **ReadNetworkTime64** function reads the 64-bit time value returned by the server using the handle returned by the **QueryNetworkTime** function. To obtain the time from a server synchronously, use the **GetNetworkTime64** function.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch. NTP servers return the time as the number of seconds since January 1, 1900.

To convert the 64-bit time value to a SYSTEMTIME structure, use the **ConvertNetworkTime64** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime64, FormatNetworkTime64, GetNetworkTime64, QueryNetworkTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeAttachThread Function

```
DWORD WINAPI TimeAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **TimeAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **TimeAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **TimeAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **TimeCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **TimeUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

QueryNetworkTime, ReadNetworkTime, TimeInitialize, TimeUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeDisableEvents Function

```
INT WINAPI TimeDisableEvents(
    HCLIENT hClient
);
```

The **TimeDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeEnableEvents, TimeFreezeEvents, TimeRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeDisableTrace Function

```
BOOL WINAPI TimeDisableTrace();
```

The **TimeDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeEnableEvents Function

```
INT WINAPI TimeEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **TimeEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **TimeRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
TIME_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
TIME_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TIME_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_WRITE	The client can now write data. This notification is sent after a

	connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TIME_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **TimeDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeDisableEvents, TimeFreezeEvents, TimeRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeEnableTrace Function

```
BOOL WINAPI TimeEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **TimeEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TimeDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeEventProc Function

```
VOID CALLBACK TimeEventProc(
   HCLIENT hClient,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **TimeEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **TimeRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **TimeRegisterEvent** function. The **TimeEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeDisableEvents, TimeEnableEvents, TimeFreezeEvents, TimeRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeFreezeEvents Function

```
INT WINAPI TimeFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **TimeFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeDisableEvents, TimeEnableEvents, TimeRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeGetErrorString Function

```
INT WINAPI TimeGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **TimeGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **TimeGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TimeGetLastError, TimeSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeGetLastError Function

```
DWORD WINAPI TimeGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **TimeSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **TimeGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **TimeSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those functions which call **TimeSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeGetErrorString, TimeSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeGetStatus Function

```
INT WINAPI TimeGetStatus(
   HCLIENT hClient
);
```

The **TimeGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

The **TimeGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
TIME_STATUS_IDLE	The client is current idle and not sending or receiving data.
TIME_STATUS_CONNECT	The client is establishing a connection with the server.
TIME_STATUS_READ	The client is reading data from the server.
TIME_STATUS_WRITE	The client is writing data to the server.
TIME_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

QueryNetworkTime, ReadNetworkTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeInitialize Function

```
BOOL WINAPI TimeInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **TimeInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TimeGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **TimeInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **TimeUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **TimeInitialize** or **TimeUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

GetNetworkTime, QueryNetworkTime, TimeUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeRegisterEvent Function

```
INT WINAPI TimeRegisterEvent(
   HCLIENT hClient,
   UINT nEventId,
   INETEVENTPROC lpfnEvent,
   DWORD_PTR dwParam
);
```

The **TimeRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Value	Description
TIME_EVENT_CONNECT	The connection to the server has completed.
TIME_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TIME_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TIME_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **TimeEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86

(32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

The **TimeRegisterEvent** function associates a callback function with a specific event. The event handler is an **TimeEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeDisableEvents, TimeEnableEvents, TimeEventProc, TimeFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeSetLastError Function

```
VOID WINAPI TimeSetLastError(
   DWORD dwErrorCode
);
```

The **TimeSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those functions which call **TimeSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **TimeGetLastError** function. The use of **TimeGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeGetErrorString, TimeGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TimeUninitialize Function

```
VOID WINAPI TimeUninitialize();
```

The **TimeUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **TimeInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **TimeUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **TimeUninitialize** for every successful call to **TimeInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

TimeInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

UpdateLocalTime Function

```
BOOL WINAPI UpdateLocalTime(
   DWORD dwNetworkTime
);
```

The **UpdateLocalTime** function sets the local system clock to the specified date and time.

Parameters

dwNetworkTime

The date and time the system clock should be set to, represented as the number of seconds since midnight, 1 January 1900.

Return Value

If the function is able to update the local time, it returns a non-zero value. If the specified time is invalid, or the user does not have the access rights to change the system clock, the function returns zero.

Remarks

The **UpdateLocalTime** function updates the local system clock to the specified date and time. The user must have the appropriate privileges to change the system clock, otherwise an error will be returned. This typically requires the application to execute with elevated privileges. Because Windows systems are usually configured to automatically synchronize the system clock with a time server using the Windows Time service (W32Time), most applications should not use this function.

This function is included for backwards compatibility and should not be used with new applications. The 32-bit network time value can only represent a date and time up to the year 2036. Use the **UpdateLocalTime64** function which accepts a 64-bit integer time value.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime, GetNetworkTime, QueryNetworkTime, TimeInitialize, UpdateLocalTime64

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

UpdateLocalTime64 Function

```
BOOL WINAPI UpdateLocalTime64(
   ULONGLONG ulNetworkTime
);
```

The **UpdateLocalTime64** function sets the local system clock to the specified date and time.

Parameters

ulNetworkTime

The date and time the system clock should be set to, represented as the number of seconds since midnight, 1 January 1900.

Return Value

If the function is able to update the local time, it returns a non-zero value. If the specified time is invalid, or the user does not have the access rights to change the system clock, the function returns zero.

Remarks

The **UpdateLocalTime64** function updates the local system clock to the specified date and time. The user must have the appropriate privileges to change the system clock, otherwise an error will be returned. This typically requires the application to execute with elevated privileges. Because Windows systems are usually configured to automatically synchronize the system clock with a time server using the Windows Time service (W32Time), most applications should not use this function.

It is important to note that the network time value is not the same as the UNIX time value used with the standard C runtime library. The **time()** function returns the number of seconds since January 1, 1970 which is commonly referred to as the UNIX epoch.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstimv12.lib

See Also

ConvertNetworkTime64, GetNetworkTime64, QueryNetworkTime, UpdateLocalTime, TimeInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Network Time Protocol Data Structures

- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Library

Retrieve and process the contents of a syndicated news feed.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSRSSV12.DLL
Version	12.0.3024.1286
LibID	B8830D80-2618-4202-9C9C-9C3AD511E9D0
Import Library	CSRSSV12.LIB
Dependencies	None

Overview

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

News feeds can be accessed remotely from a web server, or locally as an XML formatted text file. The source of the feed is determined by the URI scheme that is specified. If the http or https scheme is specified, then the feed is retrieved from a web server. If the file scheme is used, the feed is considered to be local and is accessed from the disk or local network. The News Feed library provides an API that enables you to open a feed by URL and iterate through each of the items in the feed or search for a specific feed item. The API also provides a function that can be used to parse a string that contains XML data in RSS format, where the feed may have been retrieved from other sources such as a database.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Functions

Function	Description
RssCloseFeed	Close the specified news feed and release memory allocated for the channel
RssDisableTrace	Disable logging of network function calls
RssEnableTrace	Enable logging of network function calls to a text file
RssFindItem	Find a specific news feed item using its unique identifier (GUID) property
RssGetErrorString	Return a description for the specified error code
RssGetFirstItem	Return information about the first item in the news feed channel
RssGetItem	Return information about the specified news feed item
RssGetItemCount	Return the number of news feed items in the channel
RssGetItemProperty	Return the value of the specified news feed item property or attribute
RssGetItemText	Return the text description of the specified news feed item
RssGetLastError	Return the last error code
RssGetNextItem	Return information about the next item in the news feed channel
RssInitialize	Initialize the library and validates the specified license key at runtime
RssOpenFeed	Open the specified news feed and return information about the channel
RssParseFeed	Parse the contents of a string and return information about the channel
RssRefreshFeed	Refresh the specified news feed, updating the items in the channel
RssSetLastError	Set the last error code
RssStoreFeed	Store the contents of the specified news feed in an XML formatted text file
RssUninitialize	Terminate use of the library by the application
RssValidateFeed	Validate the contents of the specified news feed, returning the number of items in the feed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssCloseFeed Function

```
INT WINAPI RssCloseFeed(
   HCHANNEL hChannel
);
```

The **RssCloseFeed** function closes the specified news feed and releases memory allocated for the channel.

Parameters

hChannel

Handle to the news feed channel.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssCloseFeed** function must be called whenever the application has completed processing the news feed. It is important to note that the memory allocated for the channel will be released when this function is called, which means that any data referenced in the RSSCHANNEL and RSSCHANNELITEM structures will no longer be valid and must not be used by the application after the feed has been closed.

This function can fail if the feed is currently being updated, such as when the **RssRefreshFeed** function is called. In this case, the channel handle will not be released and the application must attempt to close the feed at a later time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

RssOpenFeed RssParseFeed RssStoreFeed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssDisableTrace Function

```
BOOL WINAPI RssDisableTrace();
```

The **RssDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

RssEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssEnableTrace Function

```
BOOL WINAPI RssEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **RssEnableTrace** function enables the logging of network function calls to a text file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssFindItem Function

```
INT WINAPI RssFindItem(
   HCHANNEL hChannel,
   LPCTSTR lpszValue,
   DWORD dwOptions,
   LPRSSCHANNELITEM lpItem
);
```

The **RssFindItem** function searches for an item in the news feed channel which matches the unique identifier (GUID) value and returns information about that item.

Parameters

hChannel

Handle to the news feed channel.

lpszValue

A null-terminated string which specifies the value of the item being searched for. This value should uniquely identify the item in the feed, and this parameter cannot be an empty string or NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSS_FIND_GUID	Search the feed for items with a matching GUID property value. This is the default option, and is the only item property that is guaranteed to be unique in the feed. The search is case-sensitive, requiring that the ***lpszValue*** parameter match the property value exactly.
RSS_FIND_LINK	Search the feed for items with a matching link property value. For feeds that do not specify a GUID property, this is the recommended option for searching for an item. The search is not case-sensitive.
RSS_FIND_TITLE	Search the feed for items with a matching title. This option should not be used if you must ensure that the item returned is unique in the feed because there may be multiple items with the same title in the feed. The search is not case-sensitive.
RSS_FIND_PUBDATE	Search the feed for items with a matching publishing date. This option should not be used if you must ensure that the item returned is unique in the feed because more than one item may have the same publishing date. The format of the date string must match the standard format used with the RSS protocol and the match is not case-sensitive.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

It is recommended that you use the RSS_FIND_GUID option with news feeds that are using version 2.0 or later of the RSS specification. If the feed uses an earlier version, items may not include a GUID property. It is also possible that a feed may omit the GUID property even though it is considered a requirement for the current RSS specification. For the broadest compatibility with all news feeds, an application should not depend on being able to search for a specific news feed item by its GUID.

Only the GUID property is guaranteed to be unique in the feed. If the feed does not specify GUIDs for the news items, the application must use an alternate criteria such as the item hyperlink or publishing date. If there are multiple items that match the ***lpszValue*** value, the first matching item will be returned.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssGetFirstItem, RssGetItem, RssGetItemProperty, RssGetItemText, RssGetNextItem, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetErrorString Function

```
INT WINAPI RssGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **RssGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **RssGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssGetLastError, RssSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetFirstItem Function

```
BOOL WINAPI RssGetFirstItem(
   HCHANNEL hChannel,
   LPRSSCHANNELITEM lpItem
);
```

The **RssGetFirstItem** function returns information about the first item in the specified news feed channel.

Parameters

hChannel

Handle to the news feed channel.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the first item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, it returns a non-zero value. If the function fails, the return value is zero. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetFirstItem** function is used in conjunction with the **RssGetNextItem** function to enumerate the available items in the specified news feed channel. If this function fails, it typically indicates that the channel does not contain any valid news items or that the format of the news feed is invalid.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssFindItem, RssGetItem, RssGetItemProperty, RssGetItemText, RssGetNextItem, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetItem Function

```
BOOL WINAPI RssGetItem(
   HCHANNEL hChannel,
   UINT nItemId,
   LPRSSCHANNELITEM lpItem
);
```

The **RssGetItem** function returns information about the specified item in the news feed channel.

Parameters

hChannel

Handle to the news feed channel.

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this function will fail.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, it returns a value of zero. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItem** function is used to return information about a specific item in the news feed. If this function fails, it typically indicates that the item ID is invalid or that the feed does not contain any valid news items. The **RssGetItemCount** function can be used to determine the number of items contained in the feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssFindItem, RssGetFirstItem, RssGetItemCount, RssGetItemProperty, RssGetItemText, RssGetNextItem, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetItemCount Function

```
INT WINAPI RssGetItemCount(
   HCHANNEL hChannel
);
```

The **RssGetItemCount** function returns the number of items in the news feed channel.

Parameters

hChannel

Handle to the news feed channel.

Return Value

If the function succeeds, the return value is the number of items in the news feed. A value of zero indicates that the feed channel is empty. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItemCount** function is used to determine the number of items that are contained in the news feed channel, and therefore determine the maximum value of the item identifier which can be used to reference a specific item in the feed. This value is the same as the value specified by the *nItemCount* member of the **RSSCHANNEL** structure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

RssFindItem, RssGetItem, RssGetItemProperty, RssGetItemText, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetItemProperty Function

```
INT WINAPI RssGetItemProperty(
   HCHANNEL hChannel,
   UINT nItemId,
   DWORD dwReserved,
   LPCTSTR lpszProperty,
   LPCTSTR lpszAttribute,
   LPTSTR lpszValue,
   INT nMaxLength
);
```

The **RssGetItemProperty** function is used to return the value of a property for the specified item in the news feed channel.

Parameters

hChannel

Handle to the news feed channel.

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this function will fail.

dwReserved

A parameter reserved for future use. This value should always be zero.

lpszProperty

A null-terminated string which specifies the name of the item property. This parameter cannot point to an empty string or specify a NULL pointer.

lpszAttribute

A null-terminated string which specifies the name of an attribute for the property. If this parameter is an empty string or NULL pointer, the function will return the value of the property, rather than an attribute of the specified property.

lpszValue

A null-terminated string buffer which will contain the value of the specified item property or attribute. This string should be large enough to contain the property value. If this parameter is a NULL pointer, it will be ignored and the function will only return the length of value for the specified property.

nMaxLength

The maximum number of characters that may be copied into the property value buffer. If the value of this parameter is zero, then the ***lpszValue*** parameter is ignored and the function will only return the length of the value for the specified property.

Return Value

If the function succeeds, the return value is the length of the property value string. A return value of zero indicates that the property does not contain any value. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItemProperty** function is primarily used with custom item properties that may be used with extensions to the news feed. The standard properties for an item such as the title, link and description can be obtained using **RssGetItem** and related functions. However, if items in the feed contain custom properties that are not part of the standard RSS format, this function can be used to obtain those values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssFindItem, RssGetItem, RssGetItemText, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetItemText Function

```
INT WINAPI RssGetItemText(
   HCHANNEL hChannel,
   UINT nItemId,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **RssGetItemText** function is used to return a copy of an item's description.

Parameters

hChannel

Handle to the news feed channel.

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this function will fail.

lpszBuffer

A null-terminated string buffer which will contain the value of the item description. If this parameter is a NULL pointer, it will be ignored and the function will only return the length of the item description.

nMaxLength

The maximum number of characters that may be copied into the description buffer. If the value of this parameter is zero, then the ***lpszValue*** parameter is ignored and the function will only return the length of the item description.

Return Value

If the function succeeds, the return value is the length of the item description. A return value of zero indicates that the item does not have a description. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItemText** function is used to obtain a copy of the string that describes the specified item. Typically this is text that provides a summary of the news feed item and is used in conjunction with the item's title and hyperlink to additional content.

The content of an item description is typically either plain text or HTML formatted text. It is the responsibility of the application to display the content in a format is appropriate for the end-user.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssFindItem RssGetItem, RssGetItemProperty, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetLastError Function

```
DWORD WINAPI RssGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **RssSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **RssGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **RssSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CHANNEL or RSS_ERROR. Those functions which call **RssSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

RssGetErrorString, RssSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssGetNextItem Function

```
BOOL WINAPI RssGetNextItem(
  HCHANNEL hChannel,
  LPRSSCHANNELITEM lpItem
);
```

The **RssGetNextItem** function returns information about the next item in the specified news feed channel.

Parameters

hChannel

Handle to the news feed channel.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the next item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, it returns a non-zero value. If the function fails, the return value is zero. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetNextItem** function is used in conjunction with the **RssGetFirstItem** function to enumerate the available items in the specified news feed channel. If this function fails, it typically indicates that there are no more items in the news feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssFindItem, RssGetFirstItem, RssGetItem, RssGetItemProperty, RssGetItemText, RSSCHANNELITEM

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssInitialize Function

```
BOOL WINAPI RssInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **RssInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **RssGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **RssInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **RssUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **RssInitialize** or **RssUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssCloseFeed, RssOpenFeed, RssParseFeed, RssUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssOpenFeed Function

```
HCHANNEL WINAPI RssOpenFeed(
   LPCTSTR lpszFeedUrl,
   UINT nTimeout,
   DWORD dwOptions,
   LPRSSCHANNEL lpChannel
);
```

The **RssOpenFeed** function is used to open a news feed and return a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedUrl

A null-terminated string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the ***lpszFeedUrl*** parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSS_OPTION_NONE	No additional options are specified and the news feed is processed using relaxed rules when checking the validity of the feed. The library will attempt to automatically compensate for a feed that is malformed or does not strictly conform to the RSS standard.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the function. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is a handle to the news feed channel. If the function fails, the return value is INVALID_CHANNEL. To get extended error information, call **RssGetLastError**.

Remarks

A news feed may be local or remote, depending on the URL that is specified. If a local file name or file URL is specified for the feed, then it is opened locally and no network access is required. If an http or https URL is specified, then **RssOpenFeed** will attempt to download the feed from the server and

store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs.

Although the **RssOpenFeed** function will meet the needs of most applications, if you require more complex functionality such as retrieving the feed asynchronously in the background or event notifications for large transfers, you can use the SocketTools Hypertext Transfer Protocol API to download the news feed and then use the **RssParseFeed** function to parse the contents.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssCloseFeed, RssInitialize, RssParseFeed, RssRefreshFeed, RssStoreFeed, RssUninitialize, RssValidateFeed, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssParseFeed Function

```
HCHANNEL WINAPI RssParseFeed(
   LPCTSTR lpszFeedXml,
   DWORD dwOptions,
   LPRSSCHANNEL lpChannel
);
```

The **RssParseFeed** function is used to parse the contents of a news feed, returning a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedXml

A null-terminated string which contains the contents of the news feed. The string must contain XML formatted data that conforms to the RSS standard specification. This parameter cannot specify an empty string or a NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSS_OPTION_NONE	No additional options are specified.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the function. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is a handle to the news feed channel. If the function fails, the return value is INVALID_CHANNEL. To get extended error information, call **RssGetLastError**.

Remarks

The **RssParseFeed** function is an alternative to the **RssOpenFeed** function, enabling the application to process a news feed from alternative sources such as a database or compressed file. It is important to note that the string which contains the news feed XML must be properly formatted and conform to the RSS standard specification.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssCloseFeed, RssInitialize, RssOpenFeed, RssRefreshFeed, RssStoreFeed, RssUninitialize, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssRefreshFeed Function

```
INT WINAPI RssRefreshFeed(
   HCHANNEL hChannel,
   LPRSSCHANNEL lpChannel
);
```

The **RssRefreshFeed** function reloads the news feed and updates the items in the channel.

Parameters

hChannel

Handle to the news feed channel.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the function. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

When the **RssRefreshFeed** function is called, the news feed is reloaded from the original source and the items in the channel are updated. For news feeds that are frequently updated, the *nTimeToLive* member of the **RSSCHANNEL** structure can provide a hint to the application as to how frequently the feed should be refreshed.

If the news feed was originally opened using an http or https URL, this function will download the updated feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs. The same timeout period and options will be used as when the feed was originally opened.

The **RssRefreshFeed** function should only be used if the feed was opened using the **RssOpenFeed** function, otherwise the function will fail with an error indicating that the operation is not supported.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. The members of this structure returned by previous calls to either the **RssOpenFeed** or **RssRefreshFeed** functions will no longer be valid and should not be referenced. Likewise, the members of an **RSSCHANNELITEM** structure will no longer be valid after this function returns.

It is important that the application does not make any assumptions about the number of news items in the channel, or the content associated with those items after the **RssRefreshFeed** function has been called. For example, never assume that the number of items in the channel remains the same, or that the item IDs for each item remains the same. If you need to find a specific item in the news feed, use the **RssFindItem** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssCloseFeed, RssFindItem, RssGetItem, RssOpenFeed, RssStoreFeed, RSSCHANNEL

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssSetLastError Function

```
VOID WINAPI RssSetLastError(
   DWORD dwErrorCode
);
```

The **RssSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSS_ERROR. Those functions which call **RssSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **RssGetLastError** function. The use of **RssGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

RssGetErrorString, RssGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssStoreFeed Function

```
INT WINAPI RssStoreFeed(
   HCHANNEL hChannel,
   LPCTSTR lpszFileName,
   DWORD dwReserved
);
```

The **RssStoreFeed** function stores the contents of the news feed in an XML formatted text file.

Parameters

hChannel

Handle to the news feed channel.

lpszFileName

A null-terminated string which specifies the name of the file on the local system. The contents of the news feed will be stored in this file. If the file does not exist, it will be created; otherwise it will overwrite the contents of the file.

dwReserved

A reserved parameter for future use. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssOpenFeed, RssParseFeed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssUninitialize Function

```
VOID WINAPI RssUninitialize();
```

The **RssUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **RssInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **RssUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **RssUninitialize** for every successful call to **RssInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib

See Also

RssCloseFeed, RssInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RssValidateFeed Function

```
INT WINAPI RssValidateFeed(
   LPCTSTR lpszFeedUrl,
   UINT nTimeout,
   DWORD dwOptions,
   LPSYSTEMTIME lpstModified
);
```

The **RssValidateFeed** function is used to validate a news feed, returning the number of items in the feed and the date it was last modified.

Parameters

lpszFeedUrl

A null-terminated string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is reserved for future use an should always have a value of zero.

lpstModified

A pointer to a **SYSTEMTIME** structure which will specify the date that the feed was last modified when the function returns. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is the number of items in the news feed channel. If the function fails, the return value is RSS_ERROR. To get extended error information, call **RssGetLastError**.

Remarks

The **RssValidateFeed** function can be used to check that a news feed exists and is properly formatted. If the contents of the feed are valid, the function will return the number of items in the feed and the date that it was last modified. This can be useful for applications that want to periodically check a news feed and determine if the contents have changed.

The **SYSTEMTIME** structure that is populated by the function specifies the date when the feed was last modified. The function first checks the value of the lastBuildDate property of the feed channel. If that property is not defined, then it will use the value of the pubDate property. If neither are defined, then the structure members will have a value of zero.

The validation process imposes strict checks on the structure of the news feed and requires that it conform to the RSS specification. For example, the feed must have a title, link and description. Each item in the feed must have either a title or description, and the hyperlinks specified in the feed must be valid. If the feed XML is malformed, or a required property of the feed is invalid or missing, this function will fail.

If a news feed cannot be validated, it still may be possible to open the feed using the **RssOpenFeed**

function. By default, relaxed rules are used when parsing the contents of the feed and it does not check to ensure all required properties are defined and have valid values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrssv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RssOpenFeed, RssParseFeed, RssRefreshFeed, RssStoreFeed, SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

News Feed Data Structures

- RSSCHANNEL
- RSSCHANNELITEM
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RSSCHANNEL

The **RSSCHANNEL** structure contains information about the properties of the news feed channel.

```
typedef struct _RSSCHANNEL
{
    UINT        nItemCount;
    UINT        nTimeToLive;
    WORD        wVersionMajor;
    WORD        wVersionMinor;
    DWORD       dwFlags;
    DWORD       dwReserved;
    LPCTSTR     lpszTitle;
    LPCTSTR     lpszLink;
    LPCTSTR     lpszDescription;
    LPCTSTR     lpszCategory;
    LPCTSTR     lpszLanguage;
    LPCTSTR     lpszCopyright;
    LPCTSTR     lpszEditor;
    LPCTSTR     lpszWebmaster;
    LPCTSTR     lpszGenerator;
    LPCTSTR     lpszImageLink;
    LPCTSTR     lpszImageTitle;
    LPCTSTR     lpszImageUrl;
    SYSTEMTIME  stPublished;
    SYSTEMTIME  stLastBuild;
}  RSSCHANNEL, *LPRSSCHANNEL;
```

Members

nItemCount

An integer value which specifies number of news items in the channel.

nTimeToLive

An integer value which specifies the frequency in seconds at which the feed should be refreshed to obtain updated information. Not all feeds specify a time-to-live, in which case this member will have a value of zero.

wVersionMajor

A word value which specifies the major version number for the news feed.

wVersionMinor

A word value which specifies the minor version number for the news feed.

dwFlags

A value which specifies one or more option flags for the news feed channel. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A null-terminated string which specifies the name of the channel. If the content of the news feed corresponds to a website, this is typically the same as the title of the website. If a title has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a title.

lpszLink

A null-terminated string which specifies a URL to the website corresponding to the channel. Note

that this is not the URL of the news feed itself. Typically it is a link to the home page of the site which owns the news feed. If a link has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a valid link URL.

lpszDescription

A null-terminated string which describes the channel. This provides an overview of the news feed and the type of information that is provided. If a description of the feed has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a description.

lpszCategory

A null-terminated string which defines the category or categories that the channel belongs to. This property is optional and the category names themselves are user-defined. If a category has not been specified, this member will be NULL.

lpszLanguage

A null-terminated string which defines the language the channel is written in, using the standard language codes. This property is optional and if this member is NULL, the English language is typically presumed to be the default.

lpszCopyright

A null-terminated string which specifies a copyright notice for the content. If a copyright has not been specified, this member will be NULL.

lpszEditor

A null-terminated string which identifies the person responsible for managing the content of the news feed. If this property is defined, it is typically the name and email address of the feed editor. If an editor has not been specified, this member will be NULL.

lpszWebmaster

A null-terminated string which identifies the person responsible for technical issues related to the news feed. If this property is defined, it is typically the name and email address of a system administrator. If a webmaster has not been specified, this member will be NULL.

lpszGenerator

A null-terminated string which identifies the application that was used to create the news feed. If the application that generated the feed has not been specified, this member will be NULL.

lpszImageLink

A null-terminated string which specifies a URL to the website corresponding to the channel. In most cases, this is the same URL that is specified by the ***lpszLink*** member. If an image link has not been specified, this member will be NULL.

lpszImageTitle

A null-terminated string which identifies the image associated with the channel. This is usually a brief description of the image, and may be the same as the value specified by the ***lpszTitle*** member. If an image title has not been specified, this member will be NULL.

lpszImageUrl

A null-terminated string which specifies a URL for the image associated with the channel. An application can download this image and display it with the contents of the news feed. If an image URL has not been specified, this member will be NULL.

stPublished

The date that the news feed was published. For example, a feed that is associated with a weekly print publication may update this value once per week. Note that this is not necessarily the date that

the feed was last modified. If the channel does not specify the publish date, this structure will contain all zeroes.

stLastBuild

The date that the content of the channel was last modified. If the channel does not specify the build date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RSSCHANNELITEM

The **RSSCHANNELITEM** structure contains information about the properties of an item in a news feed channel.

```
typedef struct _RSSCHANNELITEM
{
    UINT        nItemId;
    DWORD       dwFlags;
    DWORD       dwReserved;
    LPCTSTR     lpszTitle;
    LPCTSTR     lpszLink;
    LPCTSTR     lpszText;
    LPCTSTR     lpszGuid;
    LPCTSTR     lpszAuthor;
    LPCTSTR     lpszSource;
    LPCTSTR     lpszComments;
    LPCTSTR     lpszEnclosure;
    SYSTEMTIME  stPublished;
}  RSSCHANNELITEM, *LPRSSCHANNELITEM;
```

Members

nItemId

An integer which identifies this item in the channel.

dwFlags

A value which specifies one or more option flags for the item. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A null-terminated string which specifies the title of the item. If a title for the item has not been specified, this member will be NULL.

lpszLink

A null-terminated string which specifies a URL that typically links to additional information related to the item. If a link for the item has not been specified, this member will be NULL.

lpszText

A null-terminated string which specifies a summary or description of the item. This may contain either plain text or HTML formatted text, and there is no fixed limit to the length of the text. If no text has been specified for the item, this member will be NULL.

lpszGuid

A null-terminated string which uniquely identifies the item in the channel. If this property is defined, it is guaranteed to be a unique, persistent value. It is important to note that this string does not have to be a standard GUID reference number, it can be any unique string. In many cases it is the same value as the item hyperlink specified by the ***lpszLink*** member, although an application should never depend on this behavior. If there is no unique identifier associated with the item, this member will be NULL.

lpszAuthor

A null-terminated string which identifies the author of the item. If this property is defined, it is

typically the name and email address of the person who created the content that the item links to. If the author is not specified, this member will be NULL.

lpszSource

A null-terminated string which identifies the source of the item, specified as a URL for the original news feed that contained it. This typically used to propagate credit for items that are aggregated by a third-party and re-published in their own channel. If the source is not specified, this member will be NULL.

lpszComments

A null-terminated string which specifies a URL that links to further discussion about the item. Typically this is a link to the comment area of a weblog or a forum topic specific to the item. If a comment link is not specified, this member will be NULL.

lpszEnclosure

A null-terminated string which specifies a URL that links to a file related to the item. This is similar to an attachment in an email message, however instead of the item containing the contents of the attached file, it only specifies a link to the file. Enclosures are most commonly used with podcasting where an item is linked to an audio or video file, however the link may reference any type of file. If there is no enclosure specified for the item, this member will be NULL.

stPublished

The date that the item was published. If the item does not specify the publish date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol Library

List and retrieve email messages from a mail server.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSPOPV12.DLL
Version	12.0.3024.1286
LibID	33EB8402-9A5F-4ADD-B520-9C24EC56F1A1
Import Library	CSPOPV12.LIB
Dependencies	None
Standards	RFC 1939

Overview

The Post Office Protocol (POP3) provides access to a user's new email messages on a mail server. Functions are provided for listing available messages and then retrieving those messages, storing them either in files or in memory. Once a user's messages have been downloaded to the local system, they are typically removed from the server. The library provides a complete interface for managing a user's mailbox and is typically used in conjunction with the Mail Message library, which is used to parse the messages retrieved from the server.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or

later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol Functions

Function	Description
PopAsyncConnect	Connect asynchronously to the specified server
PopAttachThread	Attach the specified client handle to another thread
PopCancel	Cancel the current blocking operation
PopChangePassword	Change the specified mail account password
PopCommand	Send a command to the server
PopConnect	Connect to the specified server
PopCreateSecurityCredentials	Allocate a structure to establish client security credentials
PopDeleteMessage	Delete the specified message from the mailbox
PopDeleteSecurityCredentials	Delete the specified client security credentials
PopDisableEvents	Disable the event notification mechanism
PopDisableTrace	Disable logging of network function calls
PopDisconnect	Disconnect from the current server
PopEnableEvents	Enable the client event notification mechanism
PopEnableTrace	Enable logging of network function calls to a file
PopEventProc	Callback function that processes events generated by the client
PopFreezeEvents	Suspend and resume event handling by the client
PopGetErrorString	Return a description for the specified error code
PopGetHeaderValue	Return the value of the specified header field
PopGetLastError	Return the last error code
PopGetMessage	Retrieve the specified message from the server
PopGetMessageCount	Return the number of messages available in the mailbox
PopGetMessageCountEx	Return the number of messages available in the mailbox
PopGetMessageHeaders	Retrieve the specified message header from the server
PopGetMessageId	Return the message ID string for the specified message
PopGetMessageSender	Return the address of the message sender
PopGetMessageSize	Return the size of the specified message
PopGetMessageUid	Return the unique identifier for the specified message
PopGetMultiLine	Return the client multi-line output flag
PopGetResultCode	Return the result code from the previous command
PopGetResultString	Return the result string from the previous command
PopGetSecurityInformation	Return security information about the current client connection

PopGetStatus	Return the current status of the client
PopGetTimeout	Return the number of seconds until an operation times out
PopGetTransferStatus	Return data transfer statistics
PopInitialize	Initialize the library and validates the specified license key at runtime
PopIsBlocking	Determine if the client is blocked, waiting for information
PopIsConnected	Determine if the client is connected to the server
PopIsReadable	Determine if data can be read from the server
PopIsWritable	Determine if data can be written to the server
PopLogin	Login to the server
PopOpenMessage	Open the specified message for reading on the server
PopRead	Read data returned by the server
PopRegisterEvent	Register an event handler for the specified event
PopReset	Reset the client and return to a command state
PopSendMessage	Send a message through the mail server
PopSetLastError	Set the last error code
PopSetMultiLine	Set the client multi-line output flag
PopSetTimeout	Set the number of seconds until an operation times out
PopStoreMessage	Store the contents of a message in the specified file
PopUninitialize	Terminate use of the library by the application
PopWrite	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopAsyncConnect Function

```
HCLIENT WINAPI PopAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **PopAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **PopConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
POP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
POP_OPTION_LINEBREAK	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when

	retrieving large messages from the server.
POP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
POP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
POP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option is the same as specifying POP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STLS command.
POP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the STLS command.
POP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
POP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
POP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
POP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is

	synchronized across multiple threads.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which is used to establish the client credentials for a secure connection to the server. The function **PopCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur. If this parameter is NULL, a blocking connection is established with the server.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **PopGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
POP_EVENT_CANCEL	The client has canceled the current operation.
POP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response

	to the command. The high word of the lParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
POP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **PopDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **PopAttachThread** function.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopConnect, PopDisconnect, PopInitialize, PopLogin, PopUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopAttachThread Function

```
DWORD WINAPI PopAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **PopAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **PopAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **PopAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **PopCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **PopUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopConnect, PopDisconnect, PopUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopCancel Function

```
INT WINAPI PopCancel(
   HCLIENT hClient
);
```

The **PopCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When the **PopCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopIsBlocking, PopReset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopChangePassword Function

```
BOOL WINAPI PopChangePassword(
   HCLIENT hClient,
   LPCTSTR lpszUserName,
   LPCTSTR lpszOldPassword,
   LPCTSTR lpszNewPassword
);
```

The **PopChangePassword** function changes the account password for the specified user.

Parameters

hClient

Handle to the client session.

lpszUserName

Pointer to a string which specifies the user name of the account who's password will be changed. It is not required that this be the same user name that was used to login to the mail server.

lpszOldPassword

Pointer to a string which specifies the current account password.

lpszNewPassword

Pointer to a string which specifies the new account password. When the function returns, the user's mailbox password will be set to this value.

Return Value

If the function succeeds, it will return a non-zero value. If the function fails, it will return zero. To get extended error information, call **PopGetLastError**.

Remarks

The **PopChangePassword** function is used to change the password associated with the specified account on the server. The function establishes a connection to a separate service running on the server, and does not use the POP3 protocol. For this function to succeed, the server must be configured to allow password changes using the "poppass" service, running on port 106.

Because passwords are sent over the network as clear text, this service is considered to be insecure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopConnect, PopLogin

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopCloseMessage Function

```
INT WINAPI PopCloseMessage(
   HCLIENT hClient
);
```

The **PopCloseMessage** function closes the current message that has been opened or created.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

If an message is being created, this function actually submits the message to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News messages conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopOpenMessage, PopRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopCommand Function

```
BOOL WINAPI PopCommand(
   HCLIENT hClient,
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **PopCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the command was successful, the function returns a non-zero value. If the command failed, the function returns zero. To get extended error information, call **PopGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetMultiLine, PopGetResultCode, PopGetResultString, PopSetMultiLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopConnect Function

```
HCLIENT WINAPI PopConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **PopConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
POP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
POP_OPTION_LINEBREAK	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when retrieving large messages from the server.
POP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.

POP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
POP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option is the same as specifying POP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STLS command.
POP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol and the STLS command.
POP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
POP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
POP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
POP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which is used to establish the client credentials for a secure connection to the server. The function **PopCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **PopGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **PopConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **PopAttachThread** function.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopDisconnect, PopInitialize, PopLogin, PopUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopCreateSecurityCredentials Function

```
BOOL WINAPI PopCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **PopCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **PopDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **PopGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **PopConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
PopCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                             0,
                             NULL,
                             NULL,
                             lpszCertStore,
                             lpszCertName,
                             NULL,
                             &lpSecCred);

hClient = PopConnect(lpszHostName,
                     POP_PORT_SECURE,
                     POP_TIMEOUT,
                     POP_OPTION_SECURE,
                     lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopConnect, PopDeleteSecurityCredentials, PopGetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopDeleteMessage Function

```
INT WINAPI PopDeleteMessage(
   HCLIENT hClient,
   UINT nMessage
);
```

The **PopDeleteMessage** function marks the specified message for deletion from the mailbox.

Parameters

hClient

Handle to the client session.

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

This function only marks the message for deletion. The message is not actually deleted until the user disconnects from the server. To prevent one or more marked messages from actually being deleted from the mailbox, call the **PopReset** function to reset the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetMessage, PopGetMessageCount, PopReset

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopDeleteSecurityCredentials Function

```
VOID WINAPI PopDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **PopDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)
    PopDeleteSecurityCredentials(&lpSecCred);

PopUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopCreateSecurityCredentials, PopUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopDisableEvents Function

```
INT WINAPI PopDisableEvents(
    HCLIENT hClient
);
```

The **PopDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopEnableEvents, PopFreezeEvents, PopRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopDisableTrace Function

```
BOOL WINAPI PopDisableTrace();
```

The **PopDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopDisconnect Function

```
INT WINAPI PopDisconnect(
   HCLIENT hClient
);
```

The **PopDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopConnect, PopUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopEnableEvents Function

```
INT WINAPI PopEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **PopEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **PopRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the ***hEventWnd*** parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDisableEvents, PopFreezeEvents, PopRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopEnableTrace Function

```
BOOL WINAPI PopEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **PopEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopEventProc Function

```
VOID CALLBACK PopEventProc(
   HCLIENT hClient,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **PopEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **PopRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **PopRegisterEvent** function. The **PopEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDisableEvents, PopEnableEvents, PopFreezeEvents, PopRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopFreezeEvents Function

```
INT WINAPI PopFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **PopFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDisableEvents, PopEnableEvents, PopRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetErrorString Function

```
INT WINAPI PopGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **PopGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **PopGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetLastError, PopSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetHeaderValue Function

```
INT WINAPI PopGetHeaderValue(
   HCLIENT hClient,
   UINT nMessageId,
   LPCTSTR lpszHeader,
   LPTSTR lpszValue,
   INT nMaxLength
);
```

The **PopGetHeaderValue** function returns the value of a header field in the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the function returns the length of the header field value. If the header field is not present in the message, the function will return a value of zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetHeaderValue** function returns the value of a header field from the specified message. This allows an application to be able to easily determine the value of a header (such as the sender, or the subject of the message) without downloading the entire header block or contents of the message.

This function uses the XTND XLST command, which is an extension to the POP3 protocol. Not all servers support the use of this command. If this command is not supported by the server, the function will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this function even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetMessageHeaders, PopGetMessageId, PopGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetLastError Function

```
DWORD WINAPI PopGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **PopSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **PopGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **PopSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those functions which call **PopSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetErrorString, PopSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessage Function

```
INT WINAPI PopGetMessage(
   HCLIENT hClient,
   LONG nMessageId,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved
);
```

The **PopGetMessage** function retrieves the specified message and copies the contents to a local buffer.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessage** function is used to retrieve an message from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the message data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the complete message has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event

notification must be enabled, either by calling **PopEnableEvents**, or by registering a callback function using the **PopRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **PopGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the message into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = PopGetMessage(hClient,
                        nMessageId,
                        &hgblBuffer,
                        &cbBuffer,
                        0);

if (nResult != POP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // message text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopEnableEvents, PopGetMessageHeaders, PopGetTransferStatus, PopRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageCount Function

```
INT WINAPI PopGetMessageCount(
    HCLIENT hClient
);
```

The **PopGetMessageCount** function returns the number of mail messages that are currently available in the mailbox.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, it returns the number of messages that are currently available. If no messages are available, either because the mailbox is empty or all of the messages have been deleted, this function will return zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDeleteMessage, PopGetHeaderValue, PopGetMessage, PopGetMessageCountEx, PopGetMessageHeaders, PopStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageCountEx Function

```
INT WINAPI PopGetMessageCountEx(
   HCLIENT hClient,
   UINT *lpnLastMessage,
   DWORD *lpdwMailboxSize
);
```

The **PopGetMessageCountEx** function returns the number of mail messages that are currently available in the mailbox.

Parameters

hClient

Handle to the client session.

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. This value will decrease as messages are deleted. If a NULL value is specified, this argument is ignored.

Return Value

If the function succeeds, it returns the number of messages that are currently available. If no messages are available, either because the mailbox is empty or all of the messages have been deleted, this function will return zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageCountEx** function returns the number of messages available in the mailbox, the last valid message number in the mailbox and the current size of the mailbox in bytes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDeleteMessage, PopGetHeaderValue, PopGetMessage, PopGetMessageHeaders, PopStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageHeaders Function

```
INT WINAPI PopGetMessageHeaders(
    HCLIENT hClient,
    LONG nMessageId,
    LPVOID lpvHeaders,
    LPDWORD lpdwLength
);
```

The **PopGetMessageHeaders** function retrieves the headers for the specified message from the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpvHeaders

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvHeaders*** parameter. If the ***lpvHeaders*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageHeaders** function is used to retrieve an message header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the ***lpvHeaders*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvHeaders*** parameter point to a global memory handle which will contain the message headers when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **PopEnableEvents**, or by registering a callback function using the **PopRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetHeaderValue, PopGetMessage, PopGetMessageId, PopOpenMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageId Function

```
INT WINAPI PopGetMessageId(
   HCLIENT hClient,
   UINT nMessageId,
   LPTSTR lpszMessageId,
   INT nMaxLength
);
```

The **PopGetMessageId** function returns the message identifier for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the length of the unique identifier string. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageId** function returns the message identifier from the Message-ID header of the specified message. The returned value is a typically a string which specifies the domain, date and timestamp for the message that is created when the message is submitted to the mail server for delivery. To obtain a unique identifier for the message in the mailbox, it is recommended that you use the **PopGetMessageUid** function instead.

This function uses the XTND XLST command to obtain the value of the "Message-ID" header field. If this command is not supported by the server, the function will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this function even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetHeaderValue, PopGetMessage, PopGetMessageHeaders, PopGetMessageSender, PopGetMessageUid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageSender Function

```
INT WINAPI PopGetMessageSender(
   HCLIENT hClient,
   UINT nMessageId,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **PopGetMessageSender** function returns the sender's address for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the function returns the length of the address. If the sender cannot be determined, the function will return a value of zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageSender** function returns the email address of the user who sent the specified message. This allows an application to be able to easily determine the sender, without downloading the entire header block or contents of the message.

This function uses the XSENDER command, which is an extension to the POP3 protocol, to determine the address of the authenticated sender of the message. If the command is not supported, or the server was unable to authenticate the sender, the function will use the XTND XLST command to obtain the value of the "From" header field. If this command is not supported by the server, the function will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this function even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetHeaderValue, PopGetMessageHeaders, PopGetMessageId, PopGetMessageUid

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageSize Function

```
DWORD WINAPI PopGetMessageSize(
    HCLIENT hClient,
    UINT nMessageId
);
```

The **PopGetMessageSize** function returns the size of the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is the size of the specified message in bytes. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetHeaderValue, PopGetMessageHeaders, PopGetMessageId, PopGetMessageSender, PopGetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMessageUid Function

```
INT WINAPI PopGetMessageUid(
   HCLIENT hClient,
   UINT nMessageId,
   LPTSTR lpszMessageUID,
   INT nMaxLength
);
```

The **PopGetMessageUid** function returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageUID

Address of a string buffer to receive the unique identifier for the specified message. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer for the message UID.

Return Value

If the function succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the function will return zero. If an error occurs, the function returns POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageUid** function returns the unique message identifier for the specified message. The returned value is a string which can be used to uniquely identify a specific message in the mailbox across multiple client sessions. This is commonly used by mail clients to determine if they have already retrieved a message from the server in a previous session. The UID can also be used as a key or component of the file name to reference the message after it has been stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetHeaderValue, PopGetMessage, PopGetMessageHeaders, PopGetMessageId, PopGetMessageSender

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetMultiLine Function

```
INT WINAPI PopGetMultiLine(
   HCLIENT hClient,
   LPBOOL lpbMultiLine
);
```

The **PopGetMultiLine** function returns the value of the client multi-line flag in the specified boolean parameter.

Parameters

hClient

Handle to the client session.

lpbMultiLine

A pointer to a boolean variable. This variable will be set to the current value of the client's internal multi-line flag.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopCommand, PopGetResultCode, PopGetResultString, PopSetMultiLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetResultCode Function

```
BOOL WINAPI PopGetResultCode(
    HCLIENT hClient
);
```

The **PopGetResultCode** function reads the result code returned by the server in response to a command. The result code is a boolean value, and indicates if the operation succeeded or failed.

Parameters

hClient

Handle to the client session.

Return Value

If the previous command was successful, the function returns a non-zero value. If the previous command failed, the function returns zero. To get extended error information, call **PopGetLastError**.

Remarks

Unlike most other Internet application protocols, the Post Office Protocol does not return numeric result codes to indicate success or failure. If a command is successful, the server will respond with the string "+OK" and this is indicated by the **PopGetResultCode** function returning a non-zero value. If the command fails, the server will respond with the string "-ERR" along with a description of the error, and this is indicated by the function returning a value of zero. The description of the error returned by the server can be obtained by calling the **PopGetResultString** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopCommand, PopGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetResultString Function

```
INT WINAPI PopGetResultString(
   HCLIENT hClient,
   LPTSTR lpszResult,
   INT nMaxLength
);
```

The **PopGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

The buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **PopGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopCommand, PopGetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetSecurityInformation Function

```
BOOL WINAPI PopGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **PopGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The ***dwSize*** member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the ***dwProtocol*** structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (PopGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopConnect, PopDisconnect, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetStatus Function

```
INT WINAPI PopGetStatus(
   HCLIENT hClient
);
```

The **PopGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
POP_STATUS_IDLE	The client is current idle and not sending or receiving data.
POP_STATUS_CONNECT	The client is establishing a connection with the server.
POP_STATUS_READ	The client is reading data from the server.
POP_STATUS_WRITE	The client is writing data to the server.
POP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopIsBlocking, PopIsConnected, PopIsReadable, PopIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetTimeout Function

```
INT WINAPI PopGetTimeout(
   HCLIENT hClient
);
```

The **PopGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopConnect, PopIsReadable, PopIsWritable, PopRead, PopSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopGetTransferStatus Function

```
INT WINAPI PopGetTransferStatus(
   HCLIENT hClient,
   LPPOPTRANSFERSTATUS lpStatus
);
```

The **PopGetTransferStatus** function returns information about the current file transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an POPTRANSFERSTATUS structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetTransferStatus** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopEnableEvents, PopGetStatus, PopRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopInitialize Function

```
BOOL WINAPI PopInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **PopInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **PopInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **PopUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **PopInitialize** or **PopUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopConnect, PopDisconnect, PopUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopIsBlocking Function

```
BOOL WINAPI PopIsBlocking(
   HCLIENT hClient
);
```

The **PopIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **PopIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopCancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopIsConnected Function

```
BOOL WINAPI PopIsConnected(
   HCLIENT hClient
);
```

The **PopIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopIsBlocking, PopIsReadable, PopIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopIsReadable Function

```
BOOL WINAPI PopIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **PopIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetStatus, PopIsBlocking, PopIsConnected, PopIsWritable, PopRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopIsWritable Function

```
BOOL WINAPI PopIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **PopIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetStatus, PopIsBlocking, PopIsConnected, PopIsReadable, PopWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopLogin Function

```
INT WINAPI PopLogin(
   HCLIENT hClient,
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **PopLogin** function authenticates the specified user in on the server. This function must be called after the connection has been established, and before attempting to retrieve messages or perform any other function on the server.

Parameters

hClient

Handle to the client session.

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Value	Description
POP_AUTH_DEFAULT	The default authentication scheme which sends the username and password as cleartext to the server. Because the user credentials are not encrypted, this method should only be used over a secure connection. This is the same as specifying POP_AUTH_PASS as the authentication method.
POP_AUTH_PASS	The username and password is sent to the server using the USER and PASS commands. This authentication method is supported by most servers and is the default authentication type. The credentials are not encrypted and this method should only be used over secure connections.
POP_AUTH_APOP	The APOP authentication method which uses an MD5 digest of the password. This method has been deprecated is not supported by all servers. It should only be used if required by legacy mail servers which do not support the SASL authentication methods.
POP_AUTH_LOGIN	This authentication type will use the LOGIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted and should only be used over a secure connection. The server must support the Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4422.
POP_AUTH_PLAIN	This authentication type will use the PLAIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted and should only be used over a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.

POP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the *lpszPassword* parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
POP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the *lpszPassword* parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.

lpszUserName

A null-terminated string which specifies the user name to be used to authenticate the current client session. For many service providers, the user name is the full email address of the user which owns the mailbox. In some cases, this may only be the portion of their email address before the domain name.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the current client session. If you are using the POP_AUTH_XOAUTH2 or POP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 bearer token provided by the mail service.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The POP_AUTH_LOGIN and POP_AUTH_PLAIN authentication methods require the mail server support the Simple Authentication and Security Layer (SASL) AUTH command as defined in RFC 5034. Most modern mail servers do support one or both of these methods, and they are generally preferred over the POP_AUTH_PASS method when possible. However, for backwards compatibility with legacy servers, the API will default to using POP_AUTH_PASS for client authentication.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The POP_AUTH_XOAUTH2 and POP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify

offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopConnect, PopGetMessage, PopGetMessageCountEx, PopInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopOpenMessage Function

```
INT WINAPI PopOpenMessage(
   HCLIENT hClient,
   UINT nMessageId,
   DWORD dwReserved
);
```

The **PopOpenMessage** function opens the specified message for reading.

Parameters

hClient

Handle to the client session.

nMessageId

Number that specifies which message to open. This value must be greater than zero. The first message in the mailbox is message one.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopOpenMessage** function is used to begin the process of reading the contents of a message from the from the server. Similar to how a file is opened and read, this method is followed by one or more calls to the **PopRead** method. When the entire contents of the message has been read, the **PopCloseMessage** method is used to close the message, completing the transaction on the server.

This is a lower-level function which enables the application to process the message as the contents are being returned by the server. In general, it is recommended that most applications use the **PopGetMessage** method instead, which provides a simpler method for retrieving the contents of a message.

It is important to note that you cannot use this function to read the partial contents of a message. Opening a message on the server begins a process where the entire message contents must be read and the message closed before the next command can be issued to the server. If you only want to obtain the headers for a message, use the **PopGetMessageHeaders** function instead.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopCloseMessage, PopGetMessage, PopGetMessageHeaders, PopIsReadable, PopRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopRead Function

```
INT WINAPI PopRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **PopRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that there is no more data available to be read. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When **PopRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetMessage, PopGetMessageHeaders, PopIsBlocking, PopIsReadable, PopOpenMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopRegisterEvent Function

```
INT WINAPI PopRegisterEvent(
   HCLIENT hClient,
   UINT nEvent,
   POPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **PopRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
POP_EVENT_CONNECT	The connection to the server has completed.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so

	that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **PopEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopRegisterEvent** function associates a callback function with a specific event. The event handler is an **PopEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a message is being retrieved. The POP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one POP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDisableEvents, PopEnableEvents, PopEventProc, PopFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopReset Function

```
BOOL WINAPI PopReset(
   HCLIENT hClient
);
```

The **PopReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**.

Remarks

This function will prevent any messages marked for deletion from actually being deleted from the mailbox. The client cannot be reset while the client is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopCancel, PopDeleteMessage, PopIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopSendMessage Function

```
INT WINAPI PopSendMessage(
   HCLIENT hClient,
   LPVOID lpMessage,
   DWORD dwMessageSize,
   DWORD dwReserved
);
```

The **PopSendMessage** function sends a message to the specified recipients.

Parameters

hClient

Handle to the client session.

lpMessage

Pointer to a buffer which contains the message to be submitted to the mail server for delivery.

dwMessageSize

The length of the buffer in bytes. This specifies the number of bytes to be written to the mail server. This value must be greater than zero.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopSendMessage** function sends a message through the POP3 server using the XTND XMIT command. The specified file must be in the standard format as described in RFC 822, with the recipient addresses specified in the To: and Cc: header fields. Some servers may support blind carbon copies by using addresses specified in a Bcc: header field, and then removing those addresses from the header before delivering the message.

Note that not all POP3 servers support this command, and it is recommended that you use the Simple Mail Transfer Protocol (SMTP) for general mail delivery purposes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetMessage, PopStoreMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopSetLastError Function

```
VOID WINAPI PopSetLastError(
   DWORD dwErrorCode
);
```

The **PopSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those functions which call **PopSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **PopGetLastError** function. The use of **PopGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopGetErrorString, PopGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopSetMultiLine Function

```
INT WINAPI PopSetMultiLine(
   HCLIENT hClient,
   BOOL bMultiLine
);
```

The **PopSetMultiLine** function sets the client multi-line flag into the specified value.

Parameters

hClient

Handle to the client session.

bMultiLine

A boolean flag which determines if the client is processing multiple lines of data as the result of a command.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

The **PopSetMultiLine** function should only be used in conjunction with the **PopCommand** function. If a command is issued which would result in multiple lines of output, the multi-line flag must be set TRUE. The multi-line flag must be set after each command, since it is reset to FALSE with each command that is sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopCommand, PopGetMultiLine, PopGetResultCode, PopGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopSetTimeout Function

```
INT WINAPI PopSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **PopSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopConnect, PopGetTimeout, PopRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopStoreMessage Function

```
INT WINAPI PopStoreMessage(
   HCLIENT hClient,
   UINT nMessageId,
   LPCTSTR lpszFileName
);
```

The **PopStoreMessage** function stores a message in the specified file.

Parameters

hClient

Handle to the client session.

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopStoreMessage** function provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

PopGetMessage, PopGetMessageHeaders, PopGetTransferStatus, PopSendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopUninitialize Function

```
VOID WINAPI PopUninitialize();
```

The **PopUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **PopInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **PopUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **PopUninitialize** for every successful call to **PopInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopDisconnect, PopInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

PopWrite Function

```
INT WINAPI PopWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **PopWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the POP_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cspopv12.lib

See Also

PopIsBlocking, PopIsWritable, PopRead, PopSendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Post Office Protocol Data Structures

- POPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

POPTRANSFERSTATUS

This structure is used by the PopGetTransferStatus function to return information about a message transfer in progress.

```
typedef struct _POPTRANSFERSTATUS
{
   UINT    nMessageId;
   DWORD   dwBytesTotal;
   DWORD   dwBytesCopied;
   DWORD   dwBytesPerSecond;
   DWORD   dwTimeElapsed;
   DWORD   dwTimeEstimated;
} POPTRANSFERSTATUS, *LPPOPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the message is being copied from the server to the local host, this is the size of the message on the server. If the message is being posted to the server, it is the size of message on the local system. If the message size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Remote Command Protocol Library

Execute commands on a server or establish an interactive terminal session.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSRSHV12.DLL
Version	12.0.3024.1286
LibID	BA9EA46F-9835-474A-90B1-535762F4E993
Import Library	CSRSHV12.LIB
Dependencies	None
Standards	RFC 1282

Overview

The Remote Command protocol is used to execute a command on a server and return the output of that command to the client. This is most commonly used with UNIX based servers, although there are implementations of remote command servers for the Windows operating system. The library supports both the rcmd and rshell remote execution protocols and provides methods which can be used to search the data stream for specific sequences of characters. This makes it extremely easy to write Windows applications which serve as light-weight client interfaces to commands being executed on a UNIX server or another Windows system. The library can also be used to establish a remote terminal session using the rlogin protocol, which is similar to the Telnet protocol.

The Remote Shell (rsh/rcmd) protocol is deprecated, insecure, and should never be used. This protocol transmits user credentials and command data in plaintext, making it vulnerable to interception, spoofing, and other network-based attacks. Support for this protocol will be removed in a future release. For secure remote command execution and interactive terminal sessions, it is recommended you use the Secure Shell library.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum

required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Remote Command Protocol Functions

Function	Description
RshAsyncExecute	Execute a command on the specified server
RshAsyncLogin	Establish an asynchronous login session with the specified server
RshAttachThread	Attach the specified client handle to another thread
RshCancel	Cancel the current blocking operation
RshDisableEvents	Disable asynchronous event notification
RshDisableTrace	Disable logging of network function calls
RshDisconnect	Disconnect from the current server
RshEnableEvents	Enable asynchronous event notification
RshEnableTrace	Enable logging of network function calls to a file
RshEventProc	Callback function that processes events generated by the client
RshExecute	Execute a command on the server
RshFreezeEvents	Suspend asynchronous event processing
RshGetErrorString	Return a description for the specified error code
RshGetLastError	Return the last error code
RshGetStatus	Return the current client status
RshGetTimeout	Return the number of seconds until an operation times out
RshInitialize	Initialize the library and validates the specified license key at runtime
RshIsBlocking	Determine if the client is blocked, waiting for information
RshIsConnected	Determine if the client is connected to the server
RshIsReadable	Determine if data can be read from the server
RshIsWritable	Determine if data can be written to the server
RshLogin	Establish a login session with the specified server
RshRead	Read data returned by the server
RshRegisterEvent	Register an event callback function
RshSearch	Search for a specific character sequence in the data stream
RshSetLastError	Set the last error code
RshSetTimeout	Set the number of seconds until an operation times out
RshUninitialize	Terminate use of the library by the application
RshWrite	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshAsyncExecute Function

```
HCLIENT WINAPI RshAsyncExecute(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommand,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **RshAsyncExecute** function is used to establish a connection with the server and execute the specified command.

The application should create a background worker thread and establish a connection by calling **RshExecute** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. One of the following values should be used:

Value	Description
RSH_PORT_REXEC	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command.
RSH_PORT_RSHELL	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSH_OPTION_RESERVEDPORT	This option specifies that a reserved port should be used to establish the connection. Reserved ports are those port numbers which are less than 1024. This option should be

	specified when connecting on the RSH_PORT_RSHELL port.
RSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
RSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpszUserName

A null-terminated string which specifies the username used to authenticate the client session.

lpszPassword

A null-terminated string which specifies the password used to authenticate the client session. This parameter is only used when connecting to the RSH_PORT_REXEC port. If the password is not required, this parameter may be NULL.

lpszCommand

A null-terminated string which specifies the command to execute on the server.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this parameter is NULL, then a synchronous (blocking) connection will be established with the server.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.

RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **RshDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **RshAttachThread** function.

Specifying the RSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshAsyncLogin, RshDisconnect, RshExecute, RshInitialize, RshLogin, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshAsyncLogin Function

```
HCLIENT WINAPI RshAsyncLogin(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwReserved,
   LPCTSTR lpszUserName,
   LPCTSTR lpszTerminal,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **RshAsyncLogin** function is used to establish a connection with the server.

The application should create a background worker thread and establish a connection by calling **RshLogin** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

lpszUserName

A null-terminated string which specifies the username used to authenticate the client session.

lpszTerminal

A null-terminated string which specifies the terminal type which the client will be identified as using during the session. If no particular terminal emulation is required, this parameter may be NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this parameter is NULL, then a synchronous (blocking) connection will be established with the server.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

This function uses host equivalence, where the client is permitted to login without requiring a password. Host equivalence must be configured by the server administrator and it is typically restricted to specific users. Note that if configured improperly, host equivalence can introduce a significant security loophole. Refer to your UNIX system documentation for more information about host equivalence and the various remote command services.

On UNIX based systems, the terminal name specified by the ***lpszTerminal*** parameter corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this parameter may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this parameter is passed as NULL pointer or an empty string, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is a using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320. This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.

xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **RshDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshAsyncExecute, RshDisconnect, RshExecute, RshInitialize, RshLogin, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshAttachThread Function

```
DWORD WINAPI RshAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **RshAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **RshAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **RshAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **RshCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **RshUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshCancel, RshAsyncExecute, RshAsyncLogin, RshDisconnect, RshExecute, RshInitialize, RshLogin, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshCancel Function

```
INT WINAPI RshCancel(
   HCLIENT hClient
);
```

The **RshCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When the **RshCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshDisableEvents Function

```
INT WINAPI RshDisableEvents(
    HCLIENT hClient
);
```

The **RshDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshEnableEvents, RshFreezeEvents, RshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshDisableTrace Function

```
BOOL WINAPI RshDisableTrace();
```

The **RshDisableTrace** function disables the logging of network function calls file.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshDisconnect Function

```
INT WINAPI RshDisconnect(
   HCLIENT hClient
);
```

The **RshDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshAsyncExecute, RshAsyncLogin, RshExecute, RshInitialize, RshLogin, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshEnableEvents Function

```
INT WINAPI RshEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **RshEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RshRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a

	connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **RshDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshDisableEvents, RshFreezeEvents, RshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshEnableTrace Function

```
BOOL WINAPI RshEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **RshEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshEventProc Function

```
VOID CALLBACK RshEventProc(
   HCLIENT hClient,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **RshEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **RshRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **RshRegisterEvent** function. The **RshEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshDisableEvents, RshEnableEvents, RshFreezeEvents, RshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshExecute Function

```
HCLIENT WINAPI RshExecute(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommand
);
```

The **RshExecute** function is used to establish a connection with the server and execute the specified command.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. One of the following values should be used:

Value	Description
RSH_PORT_REXEC	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command.
RSH_PORT_RSHELL	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
RSH_OPTION_RESERVEDPORT	This option specifies that a reserved port should be used to establish the connection. Reserved ports are those port numbers which are less than 1024. This option should be specified when connecting on the RSH_PORT_RSHELL port.
RSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an

	IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
RSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpszUserName

A null-terminated string which specifies the username used to authenticate the client session.

lpszPassword

A null-terminated string which specifies the password used to authenticate the client session. This parameter is only used when connecting to the RSH_PORT_REXEC port. If the password is not required, this parameter may be NULL.

lpszCommand

A null-terminated string which specifies the command to execute on the server.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **RshAttachThread** function.

Specifying the RSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshAsyncExecute, RshAsyncLogin, RshDisconnect, RshInitialize, RshLogin, RshSearch, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshFreezeEvents Function

```
INT WINAPI RshFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **RshFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshDisableEvents, RshEnableEvents, RshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshGetErrorString Function

```
INT WINAPI RshGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **RshGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **RshGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshGetLastError, RshSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshGetLastError Function

```
DWORD WINAPI RshGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **RshSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **RshGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **RshSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those functions which call **RshSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshGetErrorString, RshSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshGetStatus Function

```
INT WINAPI RshGetStatus(
   HCLIENT hClient
);
```

The **RshGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The **RshGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
RSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
RSH_STATUS_CONNECT	The client is establishing a connection with the server.
RSH_STATUS_READ	The client is reading data from the server.
RSH_STATUS_WRITE	The client is writing data to the server.
RSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshIsBlocking, RshIsConnected, RshIsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshGetTimeout Function

```
INT WINAPI RshGetTimeout(
   HCLIENT hClient
);
```

The **RshGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshInitialize Function

```
BOOL WINAPI RshInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **RshInitialize** function initializes the library and validates the specified license key at runtime.

The Remote Shell (rsh/rcmd) protocol is deprecated, insecure, and should never be used. This protocol transmits user credentials and command data in plaintext, making it vulnerable to interception, spoofing, and other network-based attacks. Support for this protocol will be removed in a future release. For secure remote command execution and interactive terminal sessions, it is recommended you use the Secure Shell library.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **RshGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The ***lpvReserved*** parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **RshInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **RshUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **RshInitialize** or **RshUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshAsyncExecute, RshAsyncLogin, RshDisconnect, RshExecute, RshLogin, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshIsBlocking Function

```
BOOL WINAPI RshIsBlocking(
   HCLIENT hClient
);
```

The **RshIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns TRUE. If the client is not performing a blocking operation, or the client handle is invalid, the function returns FALSE.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **RshIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshCancel, RshGetStatus, RshIsConnected, RshIsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshIsConnected Function

```
BOOL WINAPI RshIsConnected(
   HCLIENT hClient
);
```

The **RshIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshGetStatus, RshIsBlocking, RshIsReadable, RshIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshIsReadable Function

```
BOOL WINAPI RshIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **RshIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshGetStatus, RshIsBlocking, RshIsConnected, RshIsWritable, RshRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshIsWritable Function

```
BOOL WINAPI RshIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **RshIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshGetStatus, RshIsBlocking, RshIsConnected, RshIsReadable, RshWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshLogin Function

```
HCLIENT WINAPI RshLogin(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwReserved,
   LPCTSTR lpszUserName,
   LPCTSTR lpszTerminal
);
```

The **RshLogin** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

lpszUserName

A null-terminated string which specifies the username used to authenticate the client session.

lpszTerminal

A null-terminated string which specifies the terminal type which the client will be identified as using during the session. If no particular terminal emulation is required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

This function uses host equivalence, where the client is permitted to login without requiring a password. Host equivalence must be configured by the server administrator and it is typically restricted to specific users. Note that if configured improperly, host equivalence can introduce a significant security loophole. Refer to your UNIX system documentation for more information about host equivalence and the various remote command services. Note that if there is no host equivalence for the local host and the user account requires a password, the system will prompt for a password.

This protocol should not be used when connecting to a server on a public network because the user credentials are sent as unencrypted text. For secure remote command execution and interactive terminal sessions, it is recommended you use the Secure Shell (SSH) library.

On UNIX based systems, the terminal name specified by the ***lpszTerminal*** parameter corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this parameter may be ignored and the server will assume that the client

is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this parameter is passed as NULL pointer or an empty string, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is a using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320. This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshAsyncExecute, RshAsyncLogin, RshDisconnect, RshExecute, RshInitialize, RshSearch, RshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshRead Function

```
INT WINAPI RshRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **RshRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When **RshRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshIsBlocking, RshIsReadable, RshSearch, RshWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshRegisterEvent Function

```
INT WINAPI RshRegisterEvent(
   HCLIENT hClient,
   UINT nEventId,
   INETEVENTPROC lpfnEvent,
   DWORD_PTR dwParam
);
```

The **RshRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Value	Description
RSH_EVENT_CONNECT	The connection to the server has completed.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **RshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86

(32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The **RshRegisterEvent** function associates a callback function with a specific event. The event handler is an **RshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshDisableEvents, RshEnableEvents, RshEventProc, RshFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshSearch Function

```
BOOL WINAPI RshSearch(
   HCLIENT hClient,
   LPCTSTR lpszString,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved
);
```

The **RshSearch** function searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

hClient

Handle to the client session.

lpszString

A null-terminated string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpvBuffer

A buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the function returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the ***lpvBuffer*** parameter is NULL, this parameter should also be NULL.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds and the character sequences was found in the data stream, the return value is non-zero. If the function fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **RshGetLastError**.

Remarks

The **RshSearch** function searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The function collects the output from the server and stores it in the buffer specified by the ***lpvBuffer*** parameter. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

The ***lpvBuffer*** parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

RshExecute, RshIsBlocking, RshIsReadable, RshLogin, RshRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshSetLastError Function

```
VOID WINAPI RshSetLastError(
   DWORD dwErrorCode
);
```

The **RshSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those functions which call **RshSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **RshGetLastError** function. The use of **RshGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshGetErrorString, RshGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshSetTimeout Function

```
INT WINAPI RshSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **RshSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshUninitialize Function

```
VOID WINAPI RshUninitialize();
```

The **RshUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **RshInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **RshUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **RshUninitialize** for every successful call to **RshInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshDisconnect, RshInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

RshWrite Function

```
INT WINAPI RshWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **RshWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the RSH_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csrshv12.lib

See Also

RshIsBlocking, RshIsWritable, RshRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Secure Shell Protocol Library

Establish an interactive terminal session with an SSH server and execute remote commands.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSTSHV12.DLL
Version	12.0.3024.1286
LibID	758D7F52-CE33-4C44-B404-EBF1F15875E8
Import Library	CSTSHV12.LIB
Dependencies	None
Standards	RFC 4251

Remarks

The Secure Shell (SSH) protocol API is used to establish a secure connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The library provides an interface for establishing the connection and handling the standard I/O functions needed by the program.

Standard password and private key authentication methods are supported. The library will automatically select the strongest cipher suite available to both the client and server, providing support for AES-256 encryption and elliptical curve key exchance algorithms like ECDH and Curve25519. The control also supports some legacy algortithms for backwards compatibility with older servers but will only use them if no other options are available. If the server you are connecting with is based on OpenSSH, we recommend using OpenSSH 7.0 or later. Earlier versions of OpenSSH may request the use of encryption algorithms which are no longer considered secure.

The library also provides functions that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write lightweight client interfaces to applications running on the server. This library can be combined with the Terminal Emulation library to provide terminal emulation services for a standard ANSI or DEC-VT220 terminal.

Important: The Secure Shell (SSH) API provides a secure, encrypted alternative to legacy protocols such as Remote Shell (rsh/rcmd), Remote Login (rlogin), and Telnet. Unlike those older protocols, SSH uses strong authentication and end-to-end encryption to protect user credentials, command input, and output data from interception or tampering. SSH is the recommended protocol for remote command execution and interactive terminal sessions, particularly when connecting over public or untrusted networks. It ensures confidentiality and integrity of data in transit, and is widely supported on modern operating systems and network infrastructure. If you are currently using Telnet or Remote Shell in your applications, you are strongly encouraged to migrate to the Secure Shell API. These older protocols are inherently insecure and should not be used in any context where data security matters.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Secure Shell Protocol Functions

Function	Description
SshAsyncConnect	Connect asynchronously to the specified server
SshAttachThread	Attach the specified client handle to another thread
SshCancel	Cancel the current blocking operation
SshConnect	Connect to the specified server
SshControl	Send a control sequence to the server
SshCreateSecurityCredentials	Create a new security credentials structure
SshDeleteSecurityCredentials	Delete a previously created security credentials structure
SshDisableEvents	Disable asynchronous event notification
SshDisableTrace	Disable logging of network function calls
SshDisconnect	Disconnect from the current server
SshEnableEvents	Enable asynchronous event notification
SshEnableTrace	Enable logging of network function calls to a file
SshEventProc	Callback function that processes events generated by the client
SshExecute	Execute a command on a server and return the output in the specified buffer
SshFreezeEvents	Suspend asynchronous event processing
SshGetErrorString	Return a description for the specified error code
SshGetExitCode	Return the exit code from the remote program
SshGetLastError	Return the last error code
SshGetLineMode	Return the current mode used to send end-of-line character sequences
SshGetSecurityInformation	Return security information about the current client connection
SshGetStatus	Return the current client status
SshGetTimeout	Return the number of seconds until an operation times out
SshInitialize	Initialize the library and validates the specified license key at runtime
SshIsBlocking	Determine if the client is blocked, waiting for information
SshIsConnected	Determine if the client is connected to the server
SshIsReadable	Determine if data can be read from the server
SshIsWritable	Determine if data can be written to the server
SshPeek	Read data returned by the server, but do not remove it from the receive buffer
SshRead	Read data returned by the server
SshReadLine	Read a line of text from the server and return it in a string buffer
SshRegisterEvent	Register an event callback function
SshSearch	Search for a specific character sequence in the data stream

SshSetLastError	Set the last error code
SshSetLineMode	Change how end-of-line character sequences are sent to the server
SshSetTimeout	Set the number of seconds until an operation times out
SshUninitialize	Terminate use of the library by the application
SshWrite	Write data to the server
SshWriteLine	Write a line of text to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshAsyncConnect Function

```
HCLIENT WINAPI SshAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nTimeout,
   DWORD dwOptions,
   LPSSHOPTIONDATA lpOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **SshAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **SshConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A null-terminated string which specifies the name of the server to connect to. This may either be a fully-qualified domain name or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A null-terminated string which specifies the user name which will be used to authenticate the client session. This parameter must specify a valid user name and cannot be NULL or an empty string.

lpszPassword

A null-terminated string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this parameter can be NULL or an empty string.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.

SSH_OPTION_KEEPALIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for an extended period of time. The actual client idle timeout period is determined by the server configuration.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified. This option should never be used with interactive client sessions.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server. You should not use this option unless you require the command to execute without the normal shell environment. In most cases you should not use this option because it may cause commands to fail unexpectedly.
SSH_OPTION_TERMINAL	This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.
SSH_OPTION_COMMAND	This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudoterminal is created for the client. The szCommandLine member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.
SSH_OPTION_PROXYSERVER	This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-related members of the SSHOPTIONDATA structure should be set to the appropriate values.
SSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread

	which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpOptions

A pointer to a SSHOPTIONDATA structure which specifies additional information for one or more options. If no optional data is required, a NULL pointer may be specified.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function. You can use the **SshCreateSecurityCredentials** function to create this structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.

SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **SshDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create worker threads for each additional session if more than one active client session is required.

By default, the client session will be authenticated using standard password authentication. If you prefer to authenticate using a private key file, you must create a SECURITYCREDENTIALS structure which specifies the path to the private key and optionally the password used when creating it. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported. To determine how the private key was created, you can open the file in a text editor and the first line of the file will identify the format.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SshAttachThread** function.

Specifying the SSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
// Define the asynchronous event notification message ID
#define WM_CLIENT_EVENTS (WM_APP + 1)

HCLIENT hClient;
SSHOPTIONDATA sshOptions;

// Initialize the SSHOPTIONDATA structure and specify the
// command that should be executed on the server
ZeroMemory(&sshOptions, sizeof(sshOptions));
lstrcpyn(sshOptions.szCommandLine, lpszCommand, SSH_MAXCOMMANDLEN);

// Establish a connection with the SSH server

hClient = SshAsyncConnect(lpszHostName,
                          SSH_PORT_DEFAULT,
                          lpszUserName,
                          lpszPassword,
                          SSH_TIMEOUT,
                          SSH_OPTION_COMMAND,
                          &sshOptions,
                          NULL,
                          hAppWnd,
                          WM_CLIENT_EVENTS);
```

```
// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }

    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshConnect, SshCreateSecurityCredentials, SshDeleteSecurityCredentials, SshDisconnect, SshInitialize, SshUninitialize, SECURITYCREDENTIALS, SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshAttachThread Function

```
DWORD WINAPI SshAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **SshAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **SshAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **SshAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **SshCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **SshUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshConnect, SshDisconnect, SshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshCancel Function

```
INT WINAPI SshCancel(
   HCLIENT hClient
);
```

The **SshCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When the **SshCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshControl, SshIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshConnect Function

```
HCLIENT WINAPI SshConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   UINT nTimeout,
   DWORD dwOptions,
   LPSSHOPTIONDATA lpOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **SshConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A null-terminated string which specifies the name of the server to connect to. This may either be a fully-qualified domain name or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A null-terminated string which specifies the user name which will be used to authenticate the client session. This parameter must specify a valid user name and cannot be NULL or an empty string.

lpszPassword

A null-terminated string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this parameter can be NULL or an empty string.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPALIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for an extended period of time. The actual client idle timeout period is determined by the server configuration.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified. This option should never be used with interactive

	client sessions.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server. You should not use this option unless you require the command to execute without the normal shell environment. In most cases you should not use this option because it may cause commands to fail unexpectedly.
SSH_OPTION_TERMINAL	This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.
SSH_OPTION_COMMAND	This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudoterminal is created for the client. The szCommandLine member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.
SSH_OPTION_PROXYSERVER	This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-related members of the SSHOPTIONDATA structure should be set to the appropriate values.
SSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpOptions

A pointer to a SSHOPTIONDATA structure which specifies additional information for one or more options. If no optional data is required, a NULL pointer may be specified.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which specifies additional security-related

information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function. You can use the **SshCreateSecurityCredentials** function to create this structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SshGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **SshConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

By default, the client session will be authenticated using standard password authentication. If you prefer to authenticate using a private key file, you must create a SECURITYCREDENTIALS structure which specifies the path to the private key and optionally the password used when creating it. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported. To determine how the private key was created, you can open the file in a text editor and the first line of the file will identify the format.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SshAttachThread** function.

Specifying the SSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
HCLIENT hClient;
SSHOPTIONDATA sshOptions;

// Initialize the SSHOPTIONDATA structure and specify the
// command that should be executed on the server
ZeroMemory(&sshOptions, sizeof(sshOptions));
lstrcpyn(sshOptions.szCommandLine, lpszCommand, SSH_MAXCOMMANDLEN);

// Establish a connection with the SSH server

hClient = SshConnect(lpszHostName,
                     SSH_PORT_DEFAULT,
                     lpszUserName,
                     lpszPassword,
                     SSH_TIMEOUT,
                     SSH_OPTION_COMMAND,
                     &sshOptions,
```

```
                    NULL);

// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }

    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshCreateSecurityCredentials, SshDeleteSecurityCredentials, SshDisconnect, SshExecute, SshInitialize, SshUninitialize, SECURITYCREDENTIALS, SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshControl Function

```
INT WINAPI SshControl(
   HCLIENT hClient
   DWORD dwControlCode,
   LPVOID lpvReserved,
   DWORD dwReserved
);
```

The **SshControl** function sends a control message to the server.

Parameters

hClient

Handle to the client session.

dwControlCode

A numeric control code which specifies the control message which should be sent to the server. This may be one of the following values:

Value	Description
SSH_CONTROL_BREAK	Sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.
SSH_CONTROL_NOOP	Sends a control message to the server, but it does not perform any operation. This is typically used by clients to prevent the server from automatically closing a session that has been idle for a long period of time.
SSH_CONTROL_EOF	Sends a control message to the server indicating that the client has finished sending data. Note that this option is normally not used with interactive terminal sessions, and should only be used when required by the server.
SSH_CONTROL_PING	Sends a control message to the server which is used to test whether or not the server is responsive to the client. This is typically used by clients to attempt to detect if the connection to the server is still active.
SSH_CONTROL_REKEY	Sends a control message to the server requesting that the key exchange be performed again. This control code is ignored for SSH 1.0 sessions.

lpvReserved

A reserved parameter which should always be specified as NULL.

dwReserved

A reserved parameter which should always be specified as a value of 0.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshControl** function enables an application to send control messages to the server, which can cause it to take specific actions such as simulate a terminal break or request that the key exchanged be performed again. Some control messages are not supported by the SSH 1.0 protocol, in which case the control message is ignored.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshCancel, SshIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshCreateSecurityCredentials Function

```
BOOL WINAPI SshCreateSecurityCredentials(
   DWORD dwProtocol,
   DWORD dwOptions,
   LPCTSTR lpszKeyFile,
   LPCTSTR lpszPassword,
   LPVOID lpvReserved,
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **SshCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols.

Value	Description
SECURITY_PROTOCOL_SSH2	Version 2.0 of the protocol will be used when establishing the connection. Currently, this is the only supported protocol option for SSH connections. In older versions of SocketTools, you had the option of selecting either version 1.0 or 2.0; however, version 1.0 is no longer considered secure and is not supported.

dwOptions

This parameter is reserved for future use and should always be a value of zero.

lpszKeyFile

A null-terminated string which specifies the name of a private key file that used when authenticating the client connection. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported.

lpszPassword

A null-terminated string which specifies the password for the private key file. A value of NULL specifies that no password is required.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **SshDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

This function is only required if you wish to authenticate the client session using a private key file instead of an account password.

The ***lpszKeyFile*** path can use environment variables enclosed in percent symbols, and the path to the

private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results with multithreaded applications because the current working directory for a process is a global value and it can be changed by any thread at any time.

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **SshConnect** function.

It is safe to call **SshDeleteSecurityCredentials** after the connection has been established. An internal copy of the credentials are created during the connection process and the pointer to the credentials structure allocated by this function are no longer needed.

Example

```
HCLIENT hClient = INVALID_CLIENT;
LPSECURITYCREDENTIALS lpCredentials = NULL;

// Define the security credentials
SshCreateSecurityCredentials(SECURITY_PROTOCOL_SSH2,
                             SSH_OPTION_NONE,
                             _T("%USERPROFILE%\\.ssh\\id_rsa.pem"),
                             NULL,
                             &lpCredentials);

if (lpCredentials == NULL)
{
    // Something went wrong
    return;
}

// Establish a connection with the SSH server on port 22
hClient = SshConnect(lpszHostName,
                     SSH_PORT_DEFAULT,
                     lpszUserName,
                     NULL, // No password, using a private key file
                     SSH_TIMEOUT,
                     SSH_OPTION_COMMAND,
                     NULL,
                     lpCredentials);

// Delete the credentials
SshDeleteSecurityCredentials(&lpCredentials);

if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }
```

```
    return;
}
```

In this example, the path to the private key file will be expanded to an absolute path because the USERPROFILE environment variable defines the home directory for the current user. The path must resolve to a normal file which that can be read by the current process. If the file does not exist, or it does not specify a PEM formatted file which contains an RSA or OpenSSH private key, the connection will fail and the last error code will be set to `ST_ERROR_INVALID_PRIVATE_KEY`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshConnect, SshDeleteSecurityCredentials, SshGetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshDeleteSecurityCredentials Function

```
VOID WINAPI SshDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **SshDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Remarks

This function releases the memory allocated for the client credentials created by the **SshCreateSecurityCredentials** function. It is safe to call this function immediately after the connection has been established. An internal copy of the credentials are allocated during the connection process and the pointer to the credentials structure is no longer needed.

You should never call this function using a pointer to a credentials structure you have not explicitly allocated using the **SshCreateSecurityCredentials** function. Attempting to do so will corrupt the process heap and in most cases will throw an unhandled exception, causing your application to terminate.

Example

```
HCLIENT hClient = INVALID_CLIENT;
LPSECURITYCREDENTIALS lpCredentials = NULL;

// Define the security credentials using a private key file
SshCreateSecurityCredentials(SECURITY_PROTOCOL_SSH2,
                             SSH_OPTION_NONE,
                             _T("%USERPROFILE%\\.ssh\\id_rsa.pem"),
                             NULL,
                             &lpCredentials);

if (lpCredentials == NULL)
{
    // Something went wrong
    return;
}

// Establish a connection with the SSH server on port 22
hClient = SshConnect(lpszHostName,
                     SSH_PORT_DEFAULT,
                     lpszUserName,
                     NULL, // No password, using a private key file
                     SSH_TIMEOUT,
                     SSH_OPTION_COMMAND,
                     NULL,
                     lpCredentials);
```

```
// Delete the credentials
SshDeleteSecurityCredentials(&lpCredentials);

if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }

    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshCreateSecurityCredentials, SshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshDisableEvents Function

```
INT WINAPI SshDisableEvents(
    HCLIENT hClient
);
```

The **SshDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshEnableEvents, SshFreezeEvents, SshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshDisableTrace Function

```
BOOL WINAPI SshDisableTrace();
```

The **SshDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshDisconnect Function

```
INT WINAPI SshDisconnect(
   HCLIENT hClient
);
```

The **SshDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshConnect, SshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshEnableEvents Function

```
INT WINAPI SshEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **SshEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **SshRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the ***hEventWnd*** parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a

	connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshDisableEvents, SshFreezeEvents, SshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshEnableTrace Function

```
BOOL WINAPI SshEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **SshEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshEventProc Function

```
VOID CALLBACK SshEventProc(
   HCLIENT hClient,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **SshEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **SshRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **SshRegisterEvent** function. The **SshEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshDisableEvents, SshEnableEvents, SshFreezeEvents, SshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshExecute Function

```
INT WINAPI SshExecute(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword,
   LPCTSTR lpszCommandLine,
   UINT nTimeout,
   DWORD dwOptions,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **SshExecute** function executes a command on the server and returns the output in the specified buffer.

Parameters

lpszRemoteHost

A null-terminated string which specifies the name of the server. This may either be a fully-qualified domain name, or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A null-terminated string which specifies the user name which will be used to authenticate the client session.

lpszPassword

A null-terminated string which specifies the password which will be used to authenticate the client session.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPALIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is

	automatically set if the SSH_OPTION_COMMAND option has been specified.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the ***dwSize*** member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function.

Return Value

If the function succeeds, the return value is the exit code from the program that was executed on the server. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshExecute** function is used to execute a command on a server, read the output from that command and copy it into a local buffer. This function cannot be used if the connection to the server must be established through a proxy server; if a proxy server must be used, then you should use the **SshConnect** function to establish the connection, and then use either the **SshRead** or **SshReadLine** functions to read the output.

This function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the command output. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

When the command output is being read from the server, this function will automatically convert the data to match the end-of-line convention used on the Windows platform. This is useful when executing a command on a UNIX based system where the end-of-line is indicated by a single linefeed, while on Windows it is a carriage-return and linefeed pair. If the output contains embedded nulls or escape sequences, then this conversion will not be performed.

This function will cause the current thread to block until the command completes or a timeout occurs.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Execute a command on the server and return the data into block
// of global memory allocated by the GlobalAlloc function; the handle
// to this memory will be returned in the hgblBuffer parameter
nResult = SshExecute(lpszHostName,
                     SSH_PORT_DEFAULT,
                     lpszUserName,
                     lpszPassword,
                     lpszCommandLine,
                     SSH_TIMEOUT,
                     SSH_OPTION_NONE,
                     &hgblBuffer,
                     &cbBuffer,
                     NULL);

if (nResult != SSH_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshConnect, SshGetExitCode, SshRead, SshReadLine, SshWrite, SshWriteLine, SECURITYCREDENTIALS, SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshFreezeEvents Function

```
INT WINAPI SshFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **SshFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshDisableEvents, SshEnableEvents, SshRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetErrorString Function

```
INT WINAPI SshGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **SshGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **SshGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshGetLastError, SshSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetExitCode Function

```
INT WINAPI SshGetExitCode(
   HCLIENT hClient
);
```

The **SshGetExitCode** function returns the exit code for the remote session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the numeric exit code. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

This function should only be called after the command has completed and the **SshRead** function has returned a value of zero. In most cases, an exit code value of zero indicates success, while any other value indicates an error condition.

Note that the actual value is application dependent and is only meaningful in the context of that particular program. A program may choose or use exit codes in a non-standard way, such as having certain non-zero values indicate success.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetLastError Function

```
DWORD WINAPI SshGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SshSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **SshGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **SshSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those functions which call **SshSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetErrorString, SshSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetLineMode Function

```
INT WINAPI SshGetLineMode(
   HCLIENT hClient
);
```

The **SshGetLineMode** function returns the current line mode.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current line mode. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshGetLineMode** function returns an integer value that specifies how end-of-line character sequences are sent to the server. For more information about how newlines are processed by the library and the available options, refer to the **SshSetLineMode** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshSetLineMode, SshWrite, SshWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetSecurityInformation Function

```
BOOL WINAPI SshGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **SshGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The ***dwSize*** member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the ***dwProtocol*** structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Get the server fingerprint
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (SshGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.lpszFingerprint != NULL)
    {
        TCHAR szMessage[256];
        wsprintf(szMessage, _T("The fingerprint is %s",
securityInfo.lpszFingerprint);
        MessageBox(NULL, szMessage, "Connection", MB_OK);
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshConnect, SshDisconnect, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetStatus Function

```
INT WINAPI SshGetStatus(
   HCLIENT hClient
);
```

The **SshGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
SSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
SSH_STATUS_CONNECT	The client is establishing a connection with the server.
SSH_STATUS_AUTHENTICATE	The client is authenticating the session with the server.
4	SSH_STATUS_READ	The client is reading data from the server.
SSH_STATUS_WRITE	The client is writing data to the server.
SSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetExitCode, SshIsBlocking, SshIsConnected, SshIsReadable, SshIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshGetTimeout Function

```
INT WINAPI SshGetTimeout(
   HCLIENT hClient
);
```

The **SshGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshInitialize Function

```
BOOL WINAPI SshInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **SshInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **SshInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **SshUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **SshInitialize** or **SshUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshConnect, SshDisconnect, SshUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshIsBlocking Function

```
BOOL WINAPI SshIsBlocking(
   HCLIENT hClient
);
```

The **SshIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **SshIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshCancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshIsConnected Function

```
BOOL WINAPI SshIsConnected(
   HCLIENT hClient
);
```

The **SshIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshIsBlocking, SshIsReadable, SshIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshIsReadable Function

```
BOOL WINAPI SshIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **SshIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations on different versions of Windows, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetStatus, SshIsBlocking, SshIsConnected, SshIsWritable, SshRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshIsWritable Function

```
BOOL WINAPI SshIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **SshIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetStatus, SshIsBlocking, SshIsConnected, SshIsReadable, SshWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshPeek Function

```
INT WINAPI SshPeek(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **SshPeek** function reads the specified number of bytes from the server and copies them into the buffer, but it does not remove the data from the internal receive buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that there is no data available to be read at that time. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshPeek** function can be used to examine the data that is available to be read from the internal receive buffer. If there is no data in the receive buffer at that time, a value of zero is returned. It should be noted that this differs from the **SshRead** function, where a return value of zero indicates that there is no more data available to be read and the connection has been closed. The **SshPeek** function will never cause the client to block, and so may be safely used with asynchronous connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshIsReadable, SshRead, SshSearch, SshWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshRead Function

```
INT WINAPI SshRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **SshRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When **SshRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshIsReadable, SshPeek, SshReadLine, SshSearch, SshWrite, SshWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshReadLine Function

```
BOOL WINAPI SshReadLine(
   HCLIENT hClient,
   LPTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **SshReadLine** function reads up to a line of data and returns it in a string buffer.

Parameters

hClient

Handle to the client session.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The **SshReadLine** function reads data sent by the server and copies it into a specified string buffer. Unlike the **SshRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **SshReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the server. The data is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, read the data and then restore the client to asynchronous operation. If another client operation is attempted while **SshReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) client connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **SshRead** and **SshReadLine** function calls can be intermixed, however be aware that **SshRead** will consume any data that has already been buffered by the **SshReadLine** function and this may have unexpected results.

Unlike the **SshRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = SshReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = SshGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshIsReadble, SshRead, SshWrite, SshWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshRegisterEvent Function

```
INT WINAPI SshRegisterEvent(
   HCLIENT hClient,
   UINT nEvent,
   SSHEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **SshRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
SSH_EVENT_CONNECT	The connection to the server has completed.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86

(32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **SshRegisterEvent** function associates a callback function with a specific event. The event handler is an **SshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshDisableEvents, SshEnableEvents, SshEventProc, SshFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshSearch Function

```
BOOL WINAPI SshSearch(
   HCLIENT hClient,
   LPCTSTR lpszString,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved
);
```

The **SshSearch** function searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

hClient

Handle to the client session.

lpszString

A null-terminated string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpvBuffer

A buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the function returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the ***lpvBuffer*** parameter is NULL, this parameter should also be NULL.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds and the character sequences was found in the data stream, the return value is non-zero. If the function fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The **SshSearch** function searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server. The function collects the output from the server and stores it in the buffer specified by the ***lpvBuffer*** parameter. When the function returns, the buffer will contain everything sent by the server up to and including the matching string.

The ***lpvBuffer*** parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
LPCTSTR lpszCommand = _T("/bin/ls -l\r\n");
LPCTSTR lpszPrompt = _T("$ ");
HGLOBAL hgblOutput = NULL;
DWORD cbOutput = 0;
BOOL bResult;

// Search for a command prompt issued by the server

bResult = SshSearch(hClient,
                    lpszPrompt,
                    NULL,
                    NULL,
                    0);

// If the shell prompt was found, issue the command
// and capture the output into the hgblBuffer global
// memory buffer; the cbBuffer variable will contain
// the actual number of bytes in the buffer when the
// function returns

if (bResult)
{
    SshWrite(hClient,
            (LPBYTE)lpszCommand,
            lstrlen(lpszCommand));

    bResult = SshSearch(hClient,
                        lpszPrompt,
                        &hgblOutput,
                        &cbOutput,
                        0);
}

// Write the contents of the output buffer to the
// standard output stream

if (bResult)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    if (lpBuffer)
        fwrite(lpBuffer, 1, cbBuffer, stdout);

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshIsBlocking, SshIsReadable, SshPeek, SshRead, SshReadLine, SshWrite, SshWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshSetLastError Function

```
VOID WINAPI SshSetLastError(
   DWORD dwErrorCode
);
```

The **SshSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those functions which call **SshSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **SshGetLastError** function. The use of **SshGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetErrorString, SshGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshSetLineMode Function

```
INT WINAPI SshSetLineMode(
   HCLIENT hClient,
   INT nLineMode
);
```

The **SshSetLineMode** function changes the current line mode for the client session.

Parameters

hClient

Handle to the client session.

nLineMode

An integer value which specifies how the newlines are sent by the library. It must be one of the following values:

Value	Description
SSH_NEWLINE_DEFAULT	There are no changes to how data is sent to the server. Any carriage return or linefeed characters that are sent using the **SshWrite** function will be sent as-is. The **SshWriteLine** function will terminate each line of text with a carriage return and linefeed (CRLF) sequence. This is the default line mode that is set when a new connection is established.
SSH_NEWLINE_CR	A carriage return is used as the end-of-line character. Any data sent using the **SshWrite** function that contains only a linefeed (LF) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a carriage return (CR) character. The **SshWriteLine** function will terminate each line of text with a single carriage return character.
SSH_NEWLINE_LF	A linefeed is used as the end-of-line character. Any data sent using the **SshWrite** function that contains only a carriage return (CR) character or a carriage return an linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a linefeed (LF) character. The **SshWriteLine** function will terminate each line of text with a single linefeed character.
SSH_NEWLINE_CRLF	A carriage return and linefeed (CRLF) character sequence is used to indicate the end-of-line. Any data sent using the **SshWrite** function that contains only a carriage return (CR) or linefeed (LF) will be replaced by a carriage return and linefeed. The **SshWriteLine** function will terminate each line of text with a carriage return and linefeed sequence.

Return Value

If the function succeeds, the return value is the previous line mode for the client session. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When a connection is initially established with the server, it determines what characters are used to

indicate the end-of-line and how they are displayed. On UNIX based systems, this is controlled by the settings for the pseudo-terminal that is allocated for the client session, and can be changed using the **stty** command. In most cases, the client line mode can be left at the default. However, in some cases you may need to change the line mode, particularly if you intend to send data from a Windows text file or copied from the clipboard.

Windows uses a carriage return and linefeed (CRLF) sequence to indicate the end-of-line and a UNIX based server may interpret that as multiple newlines. To prevent this, use the **SshSetLineMode** function to change the current line mode to **SSH_NEWLINE_CR** and the CRLF sequence in the text will be replaced by a single carriage return.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetLineMode, SshWrite, SshWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshSetTimeout Function

```
INT WINAPI SshSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **SshSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshUninitialize Function

```
VOID WINAPI SshUninitialize();
```

The **SshUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **SshInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **SshUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **SshUninitialize** for every successful call to **SshInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshDisconnect, SshInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshWrite Function

```
INT WINAPI SshWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **SshWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the SSH_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib

See Also

SshIsWritable, SshRead, SshReadLine, SshSetLineMode, SshWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SshWriteLine Function

```
BOOL WINAPI SshWriteLine(
   HCLIENT hClient,
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **SshWriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

hClient

Handle to the client session.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the server. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the server.

lpnLength

A pointer to an integer value which will contain the number of characters written to the server, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The **SshWriteLine** function writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **SshWrite** function which writes arbitrary bytes of data to the server, this function is specifically designed to write a single line of text data from a string.

If the ***lpszBuffer*** string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the ***lpnLength*** parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **SshWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support sending raw Unicode data to the server. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, send the data and then

restore the client to asynchronous operation. If another network operation is attempted while **SshWriteLine** is blocked sending data to the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **SshWrite** and **SshWriteLine** function calls can be safely intermixed.

Unlike the **SshWrite** function, it is possible for data to have been written to the server if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstshv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SshIsWritable, SshRead, SshReadLine, SshSetLineMode, SshWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SSH Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO
- SSHOPTIONDATA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and fingerprint.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol was selected.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Value	Description
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.

SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

This member is not used with SSH connections and will have a value of zero.

stCertIssued

This member is not used with SSH connections and will have a value of zero.

stCertExpires

This member is not used with SSH connections and will have a value of zero.

lpszCertIssuer

This member is not used with SSH connections and will have a value of zero.

lpszCertSubject

This member is not used with SSH connections and will have a value of zero.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and cannot be used when establishing a connection with the server. Although SSH protocol version 1.0 is defined in the header file, it is important to note that it is considered insecure and is no longer supported.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A value which specifies which security protocols are supported:

Value	Description
SECURITY_PROTOCOL_SSH2	Version 2.0 of the protocol will be used when establishing the connection. Currently, this is the only supported protocol option for SSH connections. In older versions of SocketTools, you had the option of selecting either version 1.0 or 2.0; however, version 1.0 is no longer considered secure and is not supported.

dwOptions

This structure member is reserved for use with TLS connections and should always be initialized to zero.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating a server certificate. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if a private key file has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszKeyFile

A null-terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used with the SSH protocol. If the member is NULL, then no private key is used. The private key file must be in the standard PEM format defined in RFC 1422. Both the standard RSA private key format and the proprietary OpenSSH format are supported.

Remarks

This is a common structure used by several SocketTools APIs when establishing a secure connection. You should always initialize all unused members to zero or NULL and the ***dwSize*** member must be initialized to the size of the structure in bytes. In C/C++ this is done using the **sizeof** operator and other languages typically offer similar methods to determine the size of a user-defined structure.

You only need to create this structure when authenticating the client session using a private key file instead of an account password. If the private key was protected with a password, the ***lpszPassword*** member must be assigned to that value or the connection request will fail with an authentication error.

The ***lpszKeyFile*** path can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results with multithreaded applications because the current working directory for a process is a global value and it can be changed by any thread at any time.

Example

```
HCLIENT hClient;
SECURITYCREDENTIALS myCredentials;

// Initialize the SECURITYCREDENTIALS structure and specify the
// path to the private key file for the current user
ZeroMemory(&myCredentials, sizeof(myCredentials));
myCredentials.dwSize = sizeof(myCredentials);
myCredentials.lpszKeyFile = _T("%USERPROFILE%\\.ssh\\id_rsa.pem");

// Establish a connection with the SSH server on port 22
hClient = SshConnect(lpszHostName,
                     SSH_PORT_DEFAULT,
                     lpszUserName,
                     NULL, // No password, using a private key file
                     SSH_TIMEOUT,
                     SSH_OPTION_COMMAND,
                     NULL,
                     &myCredentials);
```

```
if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }

    return;
}
```

In this example, the path to the private key file will be expanded to an absolute path because the USERPROFILE environment variable defines the home directory for the current user. The path must resolve to a normal file which that can be read by the current process. If the file does not exist, or it does not specify a PEM formatted file which contains an RSA or OpenSSH private key, the connection will fail and the last error code will be set to `ST_ERROR_INVALID_PRIVATE_KEY`.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SshConnect, SshCreateSecurityCredentials, SshDeleteSecurityCredentials, SshGetSecurityInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SSHOPTIONDATA

The **SSHOPTIONDATA** structure specifies additional option information for the client session.

```
typedef struct _SSHOPTIONDATA
{
    DWORD dwSize;
    DWORD dwReserved;
    UINT  nProxyType;
    UINT  nProxyPort;
    TCHAR szProxyHost[128];
    TCHAR szProxyUser[128];
    TCHAR szProxyPassword[128];
    UINT  nTermCols;
    UINT  nTermRows;
    TCHAR szTermName[32];
    TCHAR szCommandLine[512];
} SSHOPTIONDATA, *LPSSHOPTIONDATA;
```

Members

dwSize

An unsigned integer value which specifies the size of the SSHOPTIONDATA structure. This member must be initialized prior to passing the structure to the **SshAsyncConnect** or **SshConnect** functions.

dwReserved

An unsigned integer value that is reserved for internal use, and should always be initialized to a value of zero.

nProxyType

An unsigned integer value that specifies the type of proxy that the client should connect through. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified. Possible values are:

Value	Description
SSH_PROXY_NONE	No proxy server should be used when establishing the connection.
SSH_PROXY_HTTP	The connection should be established on port 80 using HTTP. An alternate port number can be specified by setting the ***nProxyPort*** structure member to the desired value.
SSH_PROXY_TELNET	The connection should be established on port 23 using TELNET. An alternate port number can be specified by setting the ***nProxyPort*** structure member to the desired value.

nProxyPort

An unsigned integer value that specifies the port number which should be used to establish the proxy connection. A value of zero specifies that the default port number appropriate for the selected protocol should be used. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyHost

A null-terminated string which specifies the host name or IP address of the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyUser

A null-terminated string which specifies the user name which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyPassword

A null-terminated string which specifies the password which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

nTermCols

An unsigned integer value which specifies the number of columns for the virtual terminal allocated for the client session. The default number of columns is 80. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

nTermRows

An unsigned integer value which specifies the number of rows for the virtual terminal allocated for the client session. The default number of rows is 25. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szTermName

A null-terminated string which specifies the name of the terminal emulation type. On UNIX based systems, this name typically corresponds to an entry in the terminal capability database (either termcap or terminfo). If the name is not specified, then the default name terminal name of "unknown" will be used. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szCommandLine

A null-terminated string which specifies the command that should be executed on the server. The output from the command is returned to the client, and the session is terminated. This structure member is only used if the option SSH_OPTION_COMMAND has been specified.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol Library

Submit email messages for delivery to one or more recipients.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSMTPV12.DLL
Version	12.0.3024.1286
LibID	7EFD241B-CEB0-4667-8944-4231A1B46064
Import Library	CSMTPV12.LIB
Dependencies	None
Standards	RFC 821, RFC 1425, RFC 1869, RFC 2821

Overview

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The library provides an API for addressing and delivering messages, and extended features such as user authentication and delivery status notification. Unlike Microsoft's Messaging API (MAPI) or Collaboration Data Objects (CDO), there is no requirement to have certain third-party email applications installed or specific types of servers installed on the local system. The library can be used to deliver mail through a wide variety of systems, from standard UNIX based mail servers to Windows systems running Microsoft Exchange.

Using this library, messages can be delivered directly to the recipient, or they can be routed through a relay server, such as an Internet service provider's mail system. The Mail Message library can be combined with this library to provide a flexible interface for composing and delivering messages.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on

ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol Functions

Function	Description
SmtpAddRecipient	Add an address to the recipient list
SmtpAppendMessage	Append contents of specified file to the current message
SmtpAsyncConnect	Establish an asynchronous connection with a server
SmtpAsyncSendMessage	Send message to the specified recipient
SmtpAsyncSubmitMessage	Compose and submit a message for delivery to the specified mail server
SmtpAttachThread	Attach the specified client handle to another thread
SmtpAuthenticate	Authenticate the client session with a user name and password
SmtpCancel	Cancel the current blocking operation
SmtpCloseMessage	Close the message being composed and submit for delivery
SmtpCommand	Send a command to the server
SmtpConnect	Establish a connection with a server
SmtpCreateMessage	Create a new message
SmtpCreateSecurityCredentials	Allocate a structure to establish client security credentials
SmtpDeleteSecurityCredentials	Delete the specified client security credentials
SmtpDisableEvents	Disable all event notification, including event callbacks
SmtpDisableTrace	Disable logging of network function calls
SmtpDisconnect	Disconnect from the current server
SmtpEnableEvents	Enable event handling by the library
SmtpEnableTrace	Enable logging of network function calls to a file
SmtpEnumTasks	Return a list of asynchronous tasks
SmtpEventProc	Process events generated by the client
SmtpExpandAddress	Expand the specified address
SmtpFreezeEvents	Suspend and resume event handling by the client
SmtpGetCurrentDate	Return the current date and time
SmtpGetDeliveryOptions	Return the delivery options for the current session
SmtpGetErrorString	Return a description for the specified error code
SmtpGetExtendedOptions	Return the extended options supported by the server
SmtpGetLastError	Return the last error code
SmtpGetResultCode	Return the result code from the previous command
SmtpGetResultString	Return the result string from the previous command
SmtpGetSecurityInformation	Return security information about the current client connection

SmtpGetStatus	Return the current status of the client
SmtpGetTaskError	Return the last error code for the specified asynchronous task
SmtpGetTaskId	Return the unique task identifier associated with the specified client session
SmtpGetTimeout	Return the number of seconds until an operation times out
SmtpGetTransferStatus	Return data transfer statistics
SmtpInitialize	Initialize the library and validates the specified license key at runtime
SmtpIsBlocking	Determine if the client is blocked, waiting for information
SmtpIsConnected	Determine if the client is connected to the server
SmtpIsReadable	Determine if data can be read from the server
SmtpIsWritable	Determine if data can be written to the server
SmtpRegisterEvent	Register an event handler for the specified event
SmtpReset	Reset the client and return to a command state
SmtpSendMessage	Send message to the specified recipient
SmtpSetDeliveryOptions	Set the delivery options for the current session
SmtpSetLastError	Set the last error code
SmtpSetTimeout	Set the number of seconds until an operation times out
SmtpSubmitMessage	Compose and submit a message for delivery to the specified mail server
SmtpSubmitMessageEx	Compose and submit a message for delivery with additional options
SmtpTaskAbort	Abort the specified asynchronous task
SmtpTaskDone	Determine if an asynchronous task has completed
SmtpTaskResume	Resume execution of an asynchronous task
SmtpTaskSuspend	Suspend execution of an asynchronous task
SmtpTaskWait	Wait for an asynchronous task to complete
SmtpUninitialize	Terminate use of the library by the application
SmtpVerifyAddress	Verify that the specified address is valid
SmtpWrite	Write data to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAddRecipient Function

```
INT WINAPI SmtpAddRecipient(
   HCLIENT hClient,
   LPCTSTR lpszAddress
);
```

The **SmtpAddRecipient** function adds the specified address to the recipient list for the current message. This function should be called once for each recipient.

Parameters

hClient

Handle to the client session.

lpszAddress

Points to a string which specifies the address to be added to the recipient list.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpCloseMessage, SmtpCreateMessage, SmtpExpandAddress, SmtpVerifyAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAppendMessage Function

```
INT WINAPI SmtpAppendMessage(
   HCLIENT hClient,
   LPVOID lpvMessage,
   DWORD dwMessageSize,
   DWORD dwOptions
);
```

The **SmtpAppendMessage** function writes the contents of a specified file or buffer to the data stream, appending it to the current message contents.

Parameters

hClient

Handle to the client session.

lpvMessage

Pointer to a buffer which contains the message data to be appended, or a pointer to the name of the file which contains the data to be written to the data stream. The use of this parameter depends on the value of the ***dwOptions*** parameter.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

dwOptions

Specifies the source of the message data that will be written to the data stream; it may be one of the following values:

Value	Description
SMTP_MESSAGE_MEMORY	The ***lpvMessage*** parameter specifies a pointer to an array of characters. If the value of ***dwMessageSize*** is zero, then it is assumed to be a pointer to a string. If the Unicode version of this function is called and the ***dwMessageSize*** parameter is zero, the ***lpvMessage*** parameter must point to a wide string terminated with a null character. The string will be automatically converted to UTF-8 when the contents are appended to the message. In all other cases, the ***lpvMessage*** parameter is treated as a pointer to a byte array which contains the message contents and the ***dwMessageSize*** parameter specifies the number of bytes in the array.
SMTP_MESSAGE_HGLOBAL	The ***lpvMessage*** parameter specifies an HGLOBAL which contains the data to be written to the data stream. If the value of the ***dwMessageSize*** parameter is zero, then the data is assumed to be null-terminated. If the Unicode version of this function is called, the HGLOBAL handle should point to a buffer which contains a UTF-8 encoded byte array.
SMTP_MESSAGE_FILE	The ***lpvMessage*** parameter specifies a pointer to a string which contains the name of a file. The file is opened and the contents of the file are written to the data stream. The

	value of the *dwMessageSize* parameter is ignored when this option is specified.
SMTP_MESSAGE_CLIPBOARD	The *lpvMessage* and *dwMessageSize* parameters are ignored. The current contents of the clipboard are written to the data stream.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpAppendMessage** function is used to append the contents of a memory buffer, file or the system clipboard to the current message that is being composed for delivery. To send a complete RFC 822 formatted message, refer to the **SmtpSendMessage** function.

This function will cause the current thread to block until the complete message has been written, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **SmtpEnableEvents**, or by registering a callback function using the **SmtpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **SmtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpCloseMessage, SmtpCreateMessage, SmtpSendMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAsyncConnect Function

```
HCLIENT WINAPI SmtpAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszLocalName,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **SmtpAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **SmtpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For secure connections, the default port number is 465. If the secure port number is specified, an implicit TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the SMTP_OPTION_EXTENDED option must be specified.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as

	authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option is the same as specifying SMTP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the STARTTLS command. Note that the server must support secure connections using the TLS protocol.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
SMTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
SMTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has

	been specified.
SMTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpszLocalName

A null-terminated string which specifies the domain name of the local host. This parameter can be NULL or point to an empty string, in which case the local domain name is determined automatically from the system configuration. This parameter should be used if the mail server only accepts messages from a client that identifies itself using a specific domain name.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which is used to establish the client credentials for a secure connection to the server. The function **SmtpCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur. If this parameter is NULL, a blocking connection is established with the server.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SmtpGetLastError**.

Remarks

The ***lpszLocalName*** argument should only specify a domain name if it is absolutely necessary. In most cases, it is preferable to pass this parameter as NULL or an empty string and allow the library to automatically determine the correct domain name to use. Providing an invalid domain name may cause the mail server to reject the connection.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be

	posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the lParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **SmtpDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SmtpAttachThread** function.

Specifying the SMTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAuthenticate, SmtpConnect, SmtpDisconnect, SmtpInitialize, SmtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAsyncSendMessage Function

```
UINT WINAPI SmtpAsyncSendMessage(
   HCLIENT hClient,
   LPCTSTR lpszFrom,
   LPCTSTR lpszRecipient,
   LPVOID lpvMessage,
   DWORD dwMessageSize,
   DWORD dwOptions,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

Send the contents of a file or memory buffer to the specified recipients.

Parameters

hClient

Handle to the client session.

lpszFrom

Pointer to a string which specifies the email address of the sender.

lpszRecipient

Pointer to a string which specifies the recipient of the message. Multiple recipients may be specified by separating each address with a comma.

lpvMessage

Pointer to a buffer which contains the message to be delivered, or a pointer to the name of the file which contains the data to be written to the data stream. The use of this parameter depends on the value of the ***dwOptions*** parameter.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

dwOptions

Specifies the source of the message data that will be written to the data stream; it may be one of the following values:

Value	Description
SMTP_MESSAGE_MEMORY	The ***lpvMessage*** parameter specifies a pointer to an array of characters. If the value of ***dwMessageSize*** is zero, then it is assumed to be a pointer to a string.
SMTP_MESSAGE_HGLOBAL	The ***lpvMessage*** parameter specifies an HGLOBAL which contains the data to be written to the data stream. If the value of the ***dwMessageSize*** parameter is zero, then the data is assumed to be null-terminated.
SMTP_MESSAGE_FILE	The ***lpvMessage*** parameter specifies a pointer to a string which contains the name of a file. The file is opened and the contents of the file are written to the data stream. The value of the ***dwMessageSize*** parameter is ignored when this option is specified.
SMTP_MESSAGE_CLIPBOARD	The ***lpvMessage*** and ***dwMessageSize*** parameters are

	ignored. The current contents of the clipboard are written to the data stream.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SmtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpAsyncSendMessage** function is used to send the contents of a memory buffer, file or the system clipboard to the specified recipients. This function is similar to the **SmtpSendMessage** function, however it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the message is being submitted to mail server for delivery.

Because this function works asynchronously, it is important that the memory allocated for the message is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, ensure that your code does not return from the function while the message is being submitted. This can be achieved by calling the **SmtpTaskWait** function or periodically calling the **SmtpTaskDone** function to determine if the background task has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpvMessage* parameter on the heap and free that memory after the task has completed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the SMTP_EVENT_CONNECT event after the connection has been established, and the SMTP_EVENT_DISCONNECT event after the message has been submitted. This enables your application to know when the message is being submitted to the mail server, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAsyncSubmitMessage, SmtpSendMessage, SmtpSubmitMessage, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAsyncSubmitMessage Function

```
UINT WINAPI SmtpAsyncSubmitMessage(
   LPSMTPSERVER lpServer,
   LPSMTPMESSAGE lpMessage,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **SmtpSubmitMessage** function composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessage

A pointer to an **SMTPMESSAGE** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

lpEventProc

The procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, event notification is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpAsyncSubmitMessage** function provides a high-level interface that enables an application to send an email message with a single function call. This function is similar to the **SmtpSubmitMessage** function; however, it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the message is being submitted to the mail server.

The **SMTPSERVER** and **SMTPMESSAGE** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this function does not require a client session handle, and therefore it is not required that you call the **SmtpConnect** function prior to calling this function.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the SMTP_EVENT_CONNECT event after the connection has been established, and the SMTP_EVENT_DISCONNECT event after the message has been submitted. This enables your application to know when the message is being submitted to the mail server, and immediately before

the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

Example

```
SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGE mailMessage;
ZeroMemory(&mailMessage, sizeof(mailMessage));
mailMessage.lpszFrom = m_strSender;
mailMessage.lpszTo = m_strRecipients;
mailMessage.lpszSubject = m_strSubject;
mailMessage.lpszText = m_strMessage;

UINT nTaskId = SmtpAsyncSubmitMessage(&mailServer, &mailMessage, NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the message to be submitted
    SmtpTaskWait(nTaskId, INFINTE, &dwElapsed, &dwError);

    if (dwError == NO_ERROR)
        _tprintf(_T("SmtpAsyncSubmitMessage was successful\n"));
}
else
{
    DWORD dwError = SmtpGetLastError();
    _tprintf(_T("SmtpSubmitMessage failed with error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAsyncSendMessage, SmtpEventProc, SmtpSendMessage, SmtpSubmitMessage, SmtpTaskWait, SMTPMESSAGE, SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAttachThread Function

```
DWORD WINAPI SmtpAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **SmtpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **SmtpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **SmtpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **SmtpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **SmtpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpConnect, SmtpDisconnect, SmtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpAuthenticate Function

```
INT WINAPI SmtpAuthenticate(
   HCLIENT hClient,
   UINT nAuthType,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **SmtpAuthenticate** function provides client authentication information to the server.

Parameters

hClient

The handle to the client session.

nAuthType

An integer value which specifies which method the library should use to authenticate the client session. This parameter should be set to one of the following values:

Value	Description
SMTP_AUTH_LOGIN	The client will authenticate using the AUTH LOGIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. This is the default method accepted by most mail servers and is the preferred authentication type for most clients.
SMTP_AUTH_PLAIN	The client will authenticate using the AUTH PLAIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
SMTP_AUTH_XOAUTH2	The client will authenticate using the AUTH XOAUTH2 command. This authentication method does not require the user password, instead the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
SMTP_AUTH_BEARER	The client will authenticate using the AUTH OAUTHBEARER command as defined in RFC 7628. This authentication method does not require the user password, instead the ***lpszPassword*** parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.

lpszUserName

A null-terminated string which specifies the account name for the user authorized to send mail through the server.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the current client session. If you are using the SMTP_AUTH_XOAUTH2 or SMTP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 access token provided by the mail service.

Return Value

If the function succeeds, the return value is the command result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

To submit a mail message for delivery, virtually all public mail servers require clients to authenticate and will only accept messages from authorized users. In some cases, they may also require the sender email address match the account being used to authenticate the session. It is also typical for most public mail servers to reject authentication attempts over a standard (non-secure) connection. You should always use a secure connection whenever possible.

All authentication methods require the mail server to support the standard service extensions for authentication as specified in the RFC 4954. The server must support the ESMTP protocol extensions and the AUTH command. A user name and password are required for authentication. If you wish to authenticate without a user password, you must use one of the OAuth 2.0 authentication methods.

The default authentication method is SMTP_AUTH_LOGIN and this is accepted by most mail servers. It is common for mail servers to allow the SMTP_AUTH_PLAIN method as well, however it is recommended you explicitly check whether the server supports the desired authentication method by calling the **SmtpGetExtendedOptions** function. If you attempt to use an authentication method which is not supported by the server, this function will fail and the last error code will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The SMTP_AUTH_XOAUTH2 and SMTP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Example

```
BOOL bExtended = FALSE;
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server
```

```
bExtended = SmtpGetExtendedOptions(hClient, &dwOptions);

if (bUseBearerToken)
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
    {
        INT nResult = SmtpAuthenticate(hClient, SMTP_AUTH_XOAUTH2, lpszUserName,
lpszBearerToken);

        if (nResult == SMTP_ERROR)
        {
            // An error occurred during authentication; when using an
            // OAuth 2.0 bearer token, this typically means that the token
            // has expired and must be refreshed
            return;
        }
    }
    else
    {
        // The server does not support XOAUTH2
        return;
    }
}
else
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_AUTHLOGIN))
    {
        INT nResult = SmtpAuthenticate(hClient, SMTP_AUTH_LOGIN, lpszUserName,
lpszPassword);

        if (nResult == SMTP_ERROR)
        {
             // An error occurred during authentication
            return;
        }
    }
    else
    {
         // The server does not support AUTH LOGIN
        return;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpConnect, SmtpGetExtendedOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpCancel Function

```
INT WINAPI SmtpCancel(
   HCLIENT hClient
);
```

The **SmtpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

When the **SmtpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpCloseMessage Function

```
INT WINAPI SmtpCloseMessage(
    HCLIENT hClient
);
```

The **SmtpCloseMessage** function ends the composition of the current message. The server then queues the message for delivery to each recipient specified by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpCloseMessage** function should be called after all of the message data has been written to the data stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpAppendMessage, SmtpCreateMessage, SmtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpCommand Function

```
INT WINAPI SmtpCommand(
   HCLIENT hClient,
   LPCTSTR lpszCommand,
   LPCTSTR lpszParameter
);
```

The **SmtpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the command was successful, the function returns the result code. If the command failed, the function returns SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpGetResultCode, SmtpGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpConnect Function

```
HCLIENT WINAPI SmtpConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszLocalName,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **SmtpConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit TLS connections, the default port number is 465.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the SMTP_OPTION_EXTENDED option must be specified.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior

	of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
SMTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
SMTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SMTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpszLocalName

A null-terminated string which specifies the domain name of the local host. This parameter can be NULL or point to an empty string, in which case the local domain name is determined automatically from the system configuration. This parameter should be used if the mail server only accepts messages from a client that identifies itself using a specific domain name.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure which is used to establish the client credentials for a secure connection to the server. The function **SmtpCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client

credentials are not required by the server, this parameter can be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SmtpGetLastError**.

Remarks

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **SmtpConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *lpszLocalName* argument should only point to a specific domain name if it is absolutely necessary. In most cases, it is preferable to pass this parameter as NULL or an empty string and allow the library to automatically determine the correct domain name to use. Providing an invalid domain name may cause the mail server to reject the connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SmtpAttachThread** function.

Specifying the SMTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAuthenticate, SmtpDisconnect, SmtpInitialize, SmtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpCreateMessage Function

```
INT WINAPI SmtpCreateMessage(
   HCLIENT hClient
   LPCTSTR lpszSender,
   DWORD dwMessageSize,
   DWORD dwReserved
);
```

The **SmtpCreateMessage** function creates a new message for delivery.

Parameters

hClient

Handle to the client session.

lpszSender

A null-terminated string which specifies the email address of the user sending the message. This typically corresponds to the address in the From header of the message, but it is not required that they be the same.

dwMessageSize

An unsigned integer which specifies the size of the message in bytes. If the size of the message is unknown, this value should be zero. This parameter is ignored if the server does not support extended features. If the message size is larger than what the server will accept, this function will fail. Most Internet Service Providers impose a limit on the size of an email message, typically between 5 and 10 megabytes.

dwReserved

A reserved parameter. This value should be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpCreateMessage** function begins the composition of a new message to be submitted to the mail server for delivery. There are several steps that must be followed when dynamically composing a message for delivery:

1. Call the **SmtpCreateMessage** function to begin the message composition. The sender email address should generally be the same address as the one used in the "From" header field in the message.

2. Call the **SmtpAddRecipient** function for each recipient of the message. These addresses are typically specified in the "To" and "Cc" header fields in the message. Additional addresses may also be be provided which are not specified in the email message itself. This is how one or more blind carbon copies of a message is delivered. Most servers have a limit on the total number of recipients that may be specified for a single message. This limit is usually around 100 addresses.

3. Call the **SmtpWrite** function to write the contents of the message to the data stream. The application may also choose to use the **SmtpAppendMessage** function to write out a large amount of message data, or write the contents of a file to the data stream.

4. Call the **SmtpCloseMessage** function to close the message and submit it to the mail server for delivery.

For applications that do not need to dynamically compose the message and already have the message contents stored in a file or memory buffer, the **SmtpSendMessage** function is the preferred method of submitting a message for delivery.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAddRecipient, SmtpAppendMessage, SmtpCloseMessage, SmtpSendMessage, SmtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpCreateSecurityCredentials Function

```
BOOL WINAPI SmtpCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **SmtpCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **SmtpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **SmtpConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
SmtpCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                              0,
                              NULL,
                              NULL,
                              lpszCertStore,
                              lpszCertName,
                              NULL,
                              &lpSecCred);

hClient = SmtpConnect(lpszHostName,
                      SMTP_PORT_SECURE,
                      SMTP_TIMEOUT,
                      SMTP_OPTION_EXTENDED | SMTP_OPTION_SECURE,
                      lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpConnect, SmtpDeleteSecurityCredentials, SmtpGetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpDeleteSecurityCredentials Function

```
VOID WINAPI SmtpDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **SmtpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)
    SmtpDeleteSecurityCredentials(&lpSecCred);

SmtpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpCreateSecurityCredentials, SmtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpDisableEvents Function

```
INT WINAPI SmtpDisableEvents(
    HCLIENT hClient
);
```

The **SmtpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpDisableEvents** function is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this function has been called.

This function is automatically called if the client has event notification enabled, and the **SmtpDisconnect** function is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpEnableEvents, SmtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpDisableTrace Function

```
BOOL WINAPI SmtpDisableTrace();
```

The **SmtpDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpDisconnect Function

```
INT WINAPI SmtpDisconnect(
   HCLIENT hClient
);
```

The **SmtpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpConnect, SmtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpEnableEvents Function

```
INT WINAPI SmtpEnableEvents(
   HCLIENT hClient,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **SmtpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **SmtpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. To avoid conflict with standard Windows messages, this value must be greater than WM_USER (1024) or an error will be returned. If the ***hEventWnd*** parameter is NULL, this value should be WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpEnableEvents** function is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer. The client must create a window message handler, which processes the various events. The ***wParam*** argument will contain the client handle, the low word of the ***lParam*** argument will contain the event identifier, and the high word will contain any error code. If no error has occurred, the high word will have a value of zero. The following events may be generated:

Value	Description
SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be

	posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the lParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

It is not required that the client be placed in asynchronous (non-blocking) mode in order to receive event notifications, except for the connect, disconnect, read and write events. To disable event notification, call the **SmtpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpDisableEvents, SmtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpEnableTrace Function

```
BOOL WINAPI SmtpEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **SmtpEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpEnumTasks Function

```
INT WINAPI SmtpEnumTasks(
   UINT *lpTasks,
   INT nMaxTasks,
   DWORD dwOptions
);
```

Return a list of active, suspended or finished asynchronous tasks.

Parameters

lpTasks

A pointer to an array of unsigned integer values that will contain unique task identifiers when the function returns. If this parameter is NULL, the function will return the number of tasks.

nMaxTasks

An integer value that specifies the maximum number of task identifiers that may be copied into the ***lpTasks*** array. If the ***lpTasks*** parameter is NULL, this value must be zero.

dwOptions

An unsigned integer that specifies the type of asynchronous tasks that may be returned by this function. It may be a combination of the following values:

Value	Description
SMTP_TASK_DEFAULT	The list of asynchronous task IDs should include both active and suspended tasks. This option is the same as specifying both the SMTP_TASK_ACTIVE and SMTP_TASK_SUSPENDED options.
SMTP_TASK_ACTIVE	The list of asynchronous task IDs should include those tasks which are currently active. An active task represents a background connection to a server that is in the process of performing the requested action, such as uploading or downloading a file.
SMTP_TASK_SUSPENDED	The list of asynchronous task IDs should include those tasks which have been suspended. A suspended task represents a background connection that has been established, but the worker thread is not scheduled for execution.
SMTP_TASK_FINISHED	The list of asynchronous task IDs should include those tasks which have completed recently.

Return Value

If the function is successful, the return value is the number of task identifiers copied into the provided array. If there are no tasks which match the requested criteria, the return value is zero. A return value of SMTP_ERROR indicates an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpEnumTasks** function can be used to obtain a list of numeric identifiers that represent the asynchronous tasks that have been started or those that have completed. These task IDs are used by other functions to reference the background worker thread that has been created and obtain status

information for the task. For example, the **SmtpTaskDone** function can be used to determine if a particular task has completed, and the **SmtpTaskWait** function can be used to wait for a task to complete and return an error status code if the background operation failed.

There is an internal limit of 128 asynchronous tasks per process that may be active at any one time. When a task completes, the status information about that task is maintained for period of time after the task has completed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskDone, SmtpTaskResume, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpEventProc Function

```
VOID CALLBACK SmtpEventProc(
   HCLIENT hClient,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **SmtpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **SmtpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **SmtpRegisterEvent** function. This callback function is also used by asynchronous tasks to notify the application when the task has started and completed. The **SmtpEventProc** function is a placeholder for the application-defined function name.

If the callback function is invoked by an asynchronous task, it will execute in the context of the worker thread that is managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function for an asynchronous task. Controls should only be modified by the same UI thread that created their window. One common approach to resolve this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpDisableEvents, SmtpEnableEvents, SmtpFreezeEvents, SmtpRegisterEvent

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpExpandAddress Function

```
INT WINAPI SmtpExpandAddress(
   HCLIENT hClient,
   LPCTSTR lpszMailingList,
   LPTSTR lpszAddresses,
   INT nMaxLength
);
```

The **SmtpExpandAddress** function expands the specified mailing list, returning the membership of that list.

Parameters

hClient

Handle to the client session.

lpszMailingList

Points to a string which specifies the mailing list that the server should expand into full addresses.

lpszAddresses

Points to a buffer that the expanded addresses will be copied into.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpExpandAddress** function requests that the server expand the specified email address. Typically this is used to expand aliases which refer to a mailing list, returning all of the members of that list. A server may not support this command, or may restrict its usage. An application should not depend on the ability to expand addresses.

This function cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAddRecipient, SmtpVerifyAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpFreezeEvents Function

```
INT WINAPI SmtpFreezeEvents(
   HCLIENT hClient,
   BOOL bFreeze
);
```

The **SmtpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four events of the same type occur, once event handling is resumed only one of those events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpDisableEvents, SmtpEnableEvents, SmtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetCurrentDate Function

```
INT WINAPI SmtpGetCurrentDate(
   LPTSTR lpszDateString,
   INT nMaxLength
);
```

The **SmtpGetCurrentDate** function copies the current date and time to the specified buffer in a format that is commonly used in mail messages. This date format should be used in all date-related fields in the message header.

Parameters

lpszDateString

Pointer to a string buffer that will contain the current date and time when the function returns.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpCloseMessage, SmtpCreateMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetDeliveryOptions Function

```
BOOL WINAPI SmtpGetDeliveryOptions(
    HCLIENT hClient,
    LPDWORD lpdwOptions
);
```

The **SmtpGetDeliveryOptions** function returns the delivery status notification options for the current session.

Parameters

hClient

Handle to the client session.

lpdwOptions

Address of a variable that will be set to the current delivery options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Value	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetDeliveryOptions** function returns the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpConnect, SmtpGetExtendedOptions, SmtpSetDeliveryOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetErrorString Function

```
INT WINAPI SmtpGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **SmtpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **SmtpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description should be returned. If this value is zero, then the description of the last error will be returned. If the last error code is zero, indicating no error, then this function will return zero.

lpszDescription

The buffer that will contain a description of the specified error code. It is recommended that this buffer be at least 128 characters in length. If a NULL pointer is specified, then no message will be returned but the function will return the length of the error string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpGetLastError, SmtpGetResultCode, SmtpGetResultString, SmtpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetExtendedOptions Function

```
BOOL WINAPI SmtpGetExtendedOptions(
    HCLIENT hClient,
    LPDWORD lpdwOptions
);
```

The **SmtpGetExtendedOptions** function returns the extended server options for the current session.

Parameters

hClient

Handle to the client session.

lpdwOptions

Address of a variable that will be set to the current server options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Value	Description
SMTP_EXTOPT_EXPN	The server supports address expansion using the EXPN command. The **SmtpExpandAddress** function can be used to expand addresses, which typically returns the email addresses associated with a mailing list. Most public mail servers restrict or disable this functionality because it can present a security risk. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_VRFY	The server supports verification of addresses using the VRFY command. The **SmtpVerifyAddress** function can be used to verify addresses. Most public mail servers restrict the ability for clients to verify email addresses to prevent potential abuse. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_DSN	The server supports delivery status notification (DSN) which allows the sender to be notified when a message has been delivered, or when an error occurs during the delivery process. The **SmtpSetDeliveryOptions** function can be used to specify the delivery options to be used in the current session.
SMTP_EXTOPT_SIZE	The server supports the use of the SIZE parameter, which enables the client to determine the maximum message size that may be delivered through the server. Most public mail servers impose a limit of on the total size of a message, including any encoded attachments.
SMTP_EXTOPT_ETRN	The server supports the use of the ETRN command, instructing the server to start processing its message queues for a specific host. Most public mail servers do not support this capability and its use has been deprecated.
SMTP_EXTOPT_8BITMIME	The server supports the delivery of messages that contain characters with the high bit set. Most servers support this option, however it is recommended that you encode any

	message text which contains non-ASCII characters to ensure the broadest compatibility with other servers and clients.
SMTP_EXTOPT_STARTTLS	The server supports explicit TLS sessions. This extended option is used internally to determine how secure connections should be established, and if a secure connection can be made using the standard submission port.
SMTP_EXTOPT_UTF8	The server supports UTF-8 encoding in email addresses and the message envelope. Not all mail servers will have this extended capability enabled, and applications should not depend on being able to provide internationalized user and domain names unless this option bit flag has been set.

In addition, there are extended options which specify the authentication methods supported by the server. A server will typically support multiple authentication methods and may be one or more of the following values:

Value	Description
SMTP_EXTOPT_AUTHLOGIN	The server supports client authentication using the AUTH LOGIN command. This is the default authentication method and is supported by most mail servers. The user name and password are encoded in a specific format, but are not encrypted. The client should use a secure connection whenever possible.
SMTP_EXTOPT_AUTHPLAIN	The server supports client authentication using the AUTH PLAIN command. The use name and password are encoded in a specific format, but are not encrypted. If a server supports this authentication method, it is very likely it also supports AUTH LOGIN. It is recommended you use only use AUTH PLAIN authentication if the server does not support AUTH LOGIN.
SMTP_EXTOPT_XOAUTH2	The server supports client authentication using AUTH XOAUTH2 command. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method.
SMTP_EXTOPT_BEARER	The server supports client authentication using AUTH OAUTHBEARER command as specified in RFC 7628. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetExtendedOptions** function returns the extended options supported by the server. The use of extended options requires that the server support the ESMTP protocol, and that the client connect using the SMTP_OPTION_EXTENDED option.

You should check these options prior to calling **SmtpAuthenticate** to determine which authentication methods are acceptable to the server. If you wish to use an OAuth 2.0 bearer token, always check to make sure either the SMTP_EXTOPT_XOAUTH2 or SMTP_EXTOPT_BEARER bit flags are set in the options value returned by this function.

Example

```
BOOL bExtended = FALSE;
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server

bExtended = SmtpGetExtendedOptions(hClient, &dwOptions);

if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
{
    INT nResult = SmtpAuthenticate(hClient, SMTP_AUTH_XOAUTH2, lpszUserName,
lpszBearerToken);

    if (nResult == SMTP_ERROR)
    {
        // An error occurred during authentication; when using an
        // OAuth 2.0 bearer token, this typically means that the token
        // has expired and must be refreshed
        return;
    }
}
else
{
    // The server does not support XOAUTH2
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpAuthenticate, SmtpConnect, SmtpGetDeliveryOptions, SmtpSetDeliveryOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetLastError Function

```
DWORD WINAPI SmtpGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last error code value. Functions set this value by calling the **SmtpSetLastError** function. The return value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

You should call the **SmtpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **SmtpSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those functions which call **SmtpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetErrorString, SmtpSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetResultCode Function

```
INT WINAPI SmtpGetResultCode(
    HCLIENT hClient
);
```

The **SmtpGetResultCode** function reads the result code returned by the server in response to a command. The result code is an integer value, and indicates if the operation succeeded or failed.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, it returns SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The following result codes may be returned by the SMTP server:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpCommand, SmtpGetResultString

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetResultString Function

```
INT WINAPI SmtpGetResultString(
    HCLIENT hClient,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

The **SmtpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

The buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **SmtpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpCommand, SmtpGetResultCode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetSecurityInformation Function

```
BOOL WINAPI SmtpGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **SmtpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (SmtpGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpConnect, SmtpDisconnect, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetStatus Function

```
INT WINAPI SmtpGetStatus(
   HCLIENT hClient
);
```

The **SmtpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
SMTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
SMTP_STATUS_CONNECT	The client is establishing a connection with the server.
SMTP_STATUS_READ	The client is reading data from the server.
SMTP_STATUS_WRITE	The client is writing data to the server.
SMTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpIsBlocking, SmtpIsConnected, SmtpIsReadable, SmtpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetTaskError Function

```
DWORD WINAPI SmtpGetTaskError(
    UINT nTaskId
);
```

Return the last error code for the specified asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed successfully, this function returns a value of zero. A non-zero return value indicates an error has occurred.

Remarks

The **SmtpGetTaskError** function returns the last error code associated with the specified asynchronous task. If the task completed successfully, the return value will be zero. If the task is still active, the function will return the error ST_ERROR_TASK_ACTIVE. If the task has been suspended, the function will return ST_ERROR_TASK_SUSPENDED. Any other value indicates that the task completed, but the operation has failed and the error code will specify the cause of the failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskAbort, SmtpTaskResume, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetTaskId Function

```
UINT WINAPI SmtpGetTaskId(
   HCLIENT hClient
);
```

Return the asynchronous task identifier associated with the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the client handle is not associated with an asynchronous task, the function will return a value of zero.

Remarks

The **SmtpGetTaskId** function will return the task ID that is associated with a client session. This is a unique unsigned integer value that references the worker thread that was created to manage the asynchronous client session. This function should only be called within an event handler that is invoked by a background task that has been started using a function such as **SmtpAsyncGetFile**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskAbort, SmtpTaskDone, SmtpTaskResume, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetTimeout Function

```
INT WINAPI SmtpGetTimeout(
   HCLIENT hClient
);
```

The **SmtpGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpGetTransferStatus Function

```
INT WINAPI SmtpGetTransferStatus(
   HCLIENT hClient,
   LPSMTPTRANSFERSTATUS lpStatus
);
```

The **SmtpGetTransferStatus** function returns information about the message being submitted to the mail server.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an SMTPTRANSFERSTATUS structure which contains information about the status of the message being submitted for delivery.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetTransferStatus** function returns information about the current message being submitted, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no message currently being submitted, this function will return the status of the last successful submission made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain the status of a submission for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpEnableEvents, SmtpGetStatus, SmtpRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpInitialize Function

```
BOOL WINAPI SmtpInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **SmtpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **SmtpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **SmtpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **SmtpInitialize** or **SmtpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpConnect, SmtpDisconnect, SmtpUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpIsBlocking Function

```
BOOL WINAPI SmtpIsBlocking(
   HCLIENT hClient
);
```

The **SmtpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **SmtpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpCancel, SmtpIsConnected, SmtpIsReadable, SmtpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpIsConnected Function

```
BOOL WINAPI SmtpIsConnected(
    HCLIENT hClient
);
```

The **SmtpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpIsBlocking, SmtpIsReadable, SmtpIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpIsReadable Function

```
BOOL WINAPI SmtpIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **SmtpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetStatus, SmtpIsBlocking, SmtpIsConnected, SmtpIsWritable, SmtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpIsWritable Function

```
BOOL WINAPI SmtpIsWritable(
   HCLIENT hClient,
   INT nTimeout
);
```

The **SmtpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetStatus, SmtpIsBlocking, SmtpIsConnected, SmtpIsReadable, SmtpWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpRegisterEvent Function

```
INT WINAPI SmtpRegisterEvent(
   HCLIENT hClient,
   UINT nEvent,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **SmtpRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
SMTP_EVENT_CONNECT	The connection to the server has completed.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SmtpEventProc** callback

function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpRegisterEvent** function associates a callback function with a specific event. The event handler is an **SmtpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a message is being submitted to the server for delivery. The SMTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one SMTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpDisableEvents, SmtpEnableEvents, SmtpEventProc, SmtpFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpReset Function

```
INT WINAPI SmtpReset(
   HCLIENT hClient
);
```

The **SmtpReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The client cannot be reset while in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpCancel, SmtpCloseMessage, SmtpCreateMessage, SmtpIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpSendMessage Function

```
INT WINAPI SmtpSendMessage(
   HCLIENT hClient,
   LPCTSTR lpszFrom,
   LPCTSTR lpszRecipient,
   LPVOID lpvMessage,
   DWORD dwMessageSize,
   DWORD dwOptions
);
```

The **SmtpSendMessage** function sends the contents of a file or memory buffer to the specified recipients.

Parameters

hClient

Handle to the client session.

lpszFrom

Pointer to a string which specifies the email address of the sender.

lpszRecipient

Pointer to a string which specifies the recipient of the message. Multiple recipients may be specified by separating each address with a comma.

lpvMessage

Pointer to a buffer which contains the message to be delivered, or a pointer to the name of the file which contains the data to be written to the data stream. The use of this parameter depends on the value of the ***dwOptions*** parameter.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

dwOptions

Specifies the source of the message data that will be written to the data stream; it may be one of the following values:

Value	Description
SMTP_MESSAGE_MEMORY	The ***lpvMessage*** parameter specifies a pointer to an array of characters. If the value of ***dwMessageSize*** is zero, then it is assumed to be a pointer to a string. If the Unicode version of this function is called and the ***dwMessageSize*** parameter is zero, the ***lpvMessage*** parameter must point to a wide string terminated with a null character. The string will be automatically converted to UTF-8 when the message is sent. In all other cases, the ***lpvMessage*** parameter is treated as a pointer to a byte array which contains the message contents and the ***dwMessageSize*** parameter specifies the number of bytes in the array.
SMTP_MESSAGE_HGLOBAL	The ***lpvMessage*** parameter specifies an HGLOBAL which contains the data to be written to the data stream. If the value of the ***dwMessageSize*** parameter is zero, then the

	data is assumed to be null-terminated. If the Unicode version of this function is called, the HGLOBAL handle should point to a buffer which contains a UTF-8 encoded byte array.
SMTP_MESSAGE_FILE	The *lpvMessage* parameter specifies a pointer to a string which contains the name of a file. The file is opened and the contents of the file are written to the data stream. The value of the *dwMessageSize* parameter is ignored when this option is specified.
SMTP_MESSAGE_CLIPBOARD	The *lpvMessage* and *dwMessageSize* parameters are ignored. The current contents of the clipboard are written to the data stream.

Return Value

If the function succeeds, the return value is the result code from the server. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSendMessage** function is used to send the contents of a memory buffer, file or the system clipboard to the specified recipients. The message must be in the standard format as described in RFC 822 or a MIME multipart message. The MIME API can be used to compose and export a message in the correct format.

This protocol is only concerned with the delivery of a message and not its contents. Header fields in the message are not parsed to determine the recipients. This recipient parameter should be a concatenation of all recipients, including carbon copies and blind carbon copies, with each address separated with a comma.

This function will cause the current thread to block until the complete message has been delivered, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **SmtpEnableEvents**, or by registering a callback function using the **SmtpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **SmtpGetTransferStatus** function.

An alternative approach to creating a message without using the MIME API is the **SmtpSubmitMessage** function. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single function call.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAppendMessage, SmtpCloseMessage, SmtpGetTransferStatus, SmtpSubmitMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpSetDeliveryOptions Function

```
BOOL WINAPI SmtpSetDeliveryOptions(
    HCLIENT hClient,
    DWORD dwOptions
);
```

The **SmtpSetDeliveryOptions** function sets the delivery status notification options for the current session.

Parameters

hClient

Handle to the client session.

dwOptions

A bitmask that defines the current delivery options. This value is created by combining one or more of the following constants with a bitwise Or operator:

Value	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSetDeliveryOptions** function sets the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetDeliveryOptions, SmtpGetExtendedOptions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpSetLastError Function

```
VOID WINAPI SmtpSetLastError(
   DWORD dwErrorCode
);
```

The **SmtpSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the caller. A value of zero clears the last error code.

Return Value

None.

Remarks

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those functions which call **SmtpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **SmtpGetLastError** function. The use of **SmtpGetLastError** is optional. An application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetErrorString, SmtpGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpSetTimeout Function

```
INT WINAPI SmtpSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **SmtpSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpSubmitMessage Function

```
INT WINAPI SmtpSubmitMessage(
   LPSMTPSERVER lpServer,
   LPSMTPMESSAGE lpMessage,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **SmtpSubmitMessage** function composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessage

A pointer to an **SMTPMESSAGE** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

lpEventProc

The procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, event notification is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSubmitMessage** function provides a high-level interface that enables an application to send an email message with a single function call. The **SMTPSERVER** and **SMTPMESSAGE** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this function does not require a client session handle, and therefore it is not required that you call the **SmtpConnect** function prior to calling this function.

If you need to specify additional custom headers in the message that is submitted for delivery, you should use the **SmtpSubmitMessageEx** function. That version of the function uses an extended version of the message structure which will allow you to define custom headers to be included in the message.

This function will cause the calling thread to block until the message has been submitted for delivery, an error occurs or the connection to the mail server times out. If an event handler is specified, then the callback function will be periodically invoked as the message is being sent. For large messages, the SMTP_EVENT_PROGRESS event can be used to monitor the submission process and update the user interface. The **SmtpGetTransferStatus** function can be used within the callback function to obtain

information about the current status of the submission.

Example

```
SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGE mailMessage;
ZeroMemory(&mailMessage, sizeof(mailMessage));
mailMessage.lpszFrom = m_strSender;
mailMessage.lpszTo = m_strRecipients;
mailMessage.lpszSubject = m_strSubject;
mailMessage.lpszText = m_strMessage;

INT nResult = SmtpSubmitMessage(&mailServer, &mailMessage, NULL, 0);

if (nResult != SMTP_ERROR)
    _tprintf(_T("SmtpSubmitMessage was successful\n"));
else
{
    DWORD dwError = SmtpGetLastError();
    _tprintf(_T("SmtpSubmitMessage failed with error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpEventProc, SmtpSendMessage, SmtpSubmitMessageEx, SMTPMESSAGE, SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpSubmitMessageEx Function

```
INT WINAPI SmtpSubmitMessageEx(
   LPSMTPSERVER lpServer,
   LPSMTPMESSAGEEX lpMessageEx,
   DWORD dwReserved,
   SMTPEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **SmtpSubmitMessageEx** function composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessageEx

A pointer to an **SMTPMESSAGEEX** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

dwReserved

An integer value that is reserved for future use. This value must be zero or the function will fail.

lpEventProc

The procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, event notification is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSubmitMessageEx** function provides a high-level interface that enables an application to send an email message with a single function call. The **SMTPSERVER** and **SMTPMESSAGEEX** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this function does not require a client session handle, and therefore it is not required that you call the **SmtpConnect** function prior to calling this function.

This function will cause the calling thread to block until the message has been submitted for delivery, an error occurs or the connection to the mail server times out. If an event handler is specified, then the callback function will be periodically invoked as the message is being sent. For large messages, the SMTP_EVENT_PROGRESS event can be used to monitor the submission process and update the user interface. The **SmtpGetTransferStatus** function can be used within the callback function to obtain

information about the current status of the submission.

Example

```
SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGEEX mailMessageEx;
ZeroMemory(&mailMessageEx, sizeof(mailMessageEx));
mailMessageEx.dwSize = sizeof(mailMessageEx);
mailMessageEx.lpszFrom = m_strSender;
mailMessageEx.lpszTo = m_strRecipients;
mailMessageEx.lpszSubject = m_strSubject;
mailMessageEx.lpszText = m_strMessage;

INT nResult = SmtpSubmitMessageEx(&mailServer, &mailMessageEx, 0, NULL, 0);

if (nResult != SMTP_ERROR)
    _tprintf(_T("SmtpSubmitMessageEx was successful\n"));
else
{
    DWORD dwError = SmtpGetLastError();
    _tprintf(_T("SmtpSubmitMessageEx failed with error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpEventProc, SmtpSendMessage, SmtpSubmitMessage, SMTPMESSAGEEX, SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpTaskAbort Function

```
BOOL WINAPI SmtpTaskAbort(
   UINT nTaskId,
   DWORD dwMilliseconds
);
```

Abort the specified asynchronous task.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to abort.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskAbort** function signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the *dwMilliseconds* parameter has a value of zero, the function returns immediately after the background thread has been signaled. If the *dwMilliseconds* parameter is non-zero, the function will wait that amount of time for the background thread to terminate.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock. To abort a file transfer within an event handler, use the **SmtpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskDone, SmtpTaskResume, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpTaskDone Function

```
BOOL WINAPI SmtpTaskDone(
    UINT nTaskId
);
```

Determine if an asynchronous task has completed.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed, this function returns a non-zero value. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskDone** function is used to determine if the specified asynchronous task has completed. If you use this function to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this function rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **SmtpGetTaskError** function will return the last error code associated with the task. A return value of zero indicates success, while a non-zero return value specifies an error code that indicates the cause of the failure. The last error code for the task can also be retrieved using the **SmtpTaskWait** function, which causes the application to wait for the asynchronous task to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpGetTaskError, SmtpTaskAbort, SmtpTaskResume, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpTaskResume Function

```
BOOL WINAPI SmtpTaskResume(
    UINT nTaskId
);
```

Resume execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskResume** function resumes execution of the background worker thread that was previously suspended using the **SmtpTaskSuspend** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskAbort, SmtpTaskDone, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpTaskSuspend Function

```
BOOL WINAPI SmtpTaskSuspend(
    UINT nTaskId
);
```

Suspend execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskSuspend** function will suspend execution of the background worker thread associated with the task. Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **SmtpTaskResume** function. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskAbort, SmtpTaskDone, SmtpTaskResume, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpTaskWait Function

```
BOOL WINAPI SmtpTaskWait(
   UINT nTaskId,
   DWORD dwMilliseconds,
   DWORD dwReserved,
   LPDWORD lpdwElapsed,
   LPDWORD lpdwError
);
```

Wait for an asynchronous task to complete.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to complete.

dwReserved

An unsigned integer reserved for future use. This value should always be zero.

lpdwElapsed

A pointer to an unsigned integer that will contain elapsed time in milliseconds when the function returns. If this information is not required, this parameter may be NULL.

lpdwError

A pointer to an unsigned integer that will contain the error code associated with the completed task. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskWait** function waits for the specified task to complete. If the task is active and the *dwMilliseconds* parameter is non-zero, this function will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the *dwMilliseconds* parameter is zero, then this function will poll the status of the task and return immediately to the caller.

If the specified task has already completed at the time this function is called, the function will return immediately without causing the current thread to block. If the *lpdwElapsed* parameter is not NULL, it will contain the number of milliseconds that it took for the task to complete. If the *lpdwError* parameter is not NULL, it will contain the last error code value that was set by the worker thread before it terminated. If this value is zero, that means that the background operation was successful and no error occurred. A non-zero value will indicate that the background operation has failed.

You should not call this function from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this function is blocked waiting for the background task to complete, and this can cause your application to appear non-

responsive to the end-user. If you have a GUI application and you need to periodically check to see if a task has completed, create a timer to periodically call the **SmtpTaskDone** function. When it returns a non-zero value (indicating that the task has completed), you can safely call **SmtpTaskWait** to obtain the elapsed time and last error code without blocking the current thread.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpTaskDone, SmtpTaskResume, SmtpTaskSuspend, SmtpTaskWait

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpUninitialize Function

```
VOID WINAPI SmtpUninitialize();
```

The **SmtpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **SmtpInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **SmtpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **SmtpUninitialize** for every successful call to **SmtpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpConnect, SmtpDisconnect, SmtpInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpVerifyAddress Function

```
INT WINAPI SmtpVerifyAddress(
   HCLIENT hClient,
   LPCTSTR lpszAddress,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **SmtpVerifyAddress** function verifies the specified address is valid.

Parameters

hClient

Handle to the client session.

lpszAddress

Points to a string which specifies the address that the server should verify.

lpszBuffer

Points to a buffer that the verified address will be copied into.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpVerifyAddress** function requests that the server verify the specified email address. Typically this is used to verify that a recipient address is valid, and return a fully qualified email address for that recipient. A server may not support this command, or may restrict its usage. An application should not depend on the ability to verify addresses.

This function cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csmtpv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmtpAddRecipient, SmtpExpandAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmtpWrite Function

```
INT WINAPI SmtpWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **SmtpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the SMTP_EVENT_WRITE asynchronous notification message before it resumes sending data.

If the **SmtpWrite** function is used to send the message contents to the server, the application must first call the **SmtpCreateMessage** function to specify the sender and the length of the message, followed by one or more calls to the **SmtpAddRecipient** function to specify each recipient of the message. When all of the message text has been submitted to the server, the application must call the **SmtpCloseMessage** function.

The message text is filtered by the **SmtpWrite** function, and it will automatically normalize end-of-line character sequences to ensure the message meets the protocol requirements. The message itself must be in a standard RFC 822 or multi-part MIME message format, or the server may reject the message. Binary data, such as file attachments, should always be encoded. The MIME API can be used to compose and export a message in the correct format, which can then be submitted to the server.

It is recommended that most applications use the **SmtpSendMessage** function to submit the message for delivery.

An alternative approach to creating a message without using the MIME API is the **SmtpSubmitMessage** function. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single function call.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: csmtpv12.lib

See Also

SmtpAddRecipient, SmtpAppendMessage, SmtpCloseMessage, SmtpCreateMessage, SmtpSendMessage, SmtpSubmitMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Simple Mail Transfer Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO
- SMTPMESSAGE
- SMTPMESSAGEEX
- SMTPSERVER
- SMTPTRANSFERSTATUS
- SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPMESSAGE

This structure provides information about the contents of a message and is used by the SmtpSubmitMessage function.

```
typedef struct _SMTPMESSAGE
{
    LPCTSTR  lpszFrom;
    LPCTSTR  lpszTo;
    LPCTSTR  lpszCc;
    LPCTSTR  lpszBcc;
    LPCTSTR  lpszSubject;
    LPCTSTR  lpszText;
    LPCTSTR  lpszHTML;
    LPCTSTR  lpszAttach;
    UINT     nCharSet;
    UINT     nEncType;
    DWORD    dwReserved;
} SMTPMESSAGE, *LPSMTPMESSAGE;
```

Members

lpszFrom

A null-terminated string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A null-terminated string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A null-terminated string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A null-terminated string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the ***lpszTo*** and ***lpszCc*** members, any addresses specified by this member will not be included in the header of the email message.

lpszSubject

A null-terminated string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpszText

A null-terminated string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the ***lpszHTML*** member must specify the body of the message.

lpszHTML

A null-terminated string which contains the message using HTML formatting. If the ***lpszText*** member is not NULL, then a multipart message will be created with both plain text and HTML text

as the alternative. This allows mail clients to select which message body they wish to display. If the ***lpszText*** member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpszAttach

A null-terminated string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

An integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Value	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.
MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be

used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using the **SmtpSubmitMessage** function. It is required that you specify a sender, at least one recipient and a message body. All other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the structure.

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* member cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPMESSAGEEX

This structure provides information about the contents of a message and is used by the SmtpSubmitMessageEx function.

```
typedef struct _SMTPMESSAGEEX
{
    DWORD    dwSize
    LPCTSTR  lpszFrom;
    LPCTSTR  lpszTo;
    LPCTSTR  lpszCc;
    LPCTSTR  lpszBcc;
    LPCTSTR  lpszDate;
    LPCTSTR  lpszSubject;
    LPCTSTR  lpszHeaders;
    LPCTSTR  lpszText;
    LPCTSTR  lpszHTML;
    LPCTSTR  lpszAttach;
    UINT     nCharSet;
    UINT     nEncType;
    DWORD    dwReserved;
} SMTPMESSAGEEX, *LPSMTPMESSAGEEX;
```

Members

dwSize

An integer value that specifies the size of the **SMTPMESSAGEEX** data structure. This must always be explicitly defined, and if the value is incorrect, an error will be returned. This structure member is used to ensure that the correct version of the structure is being passed to the function.

lpszFrom

A null-terminated string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A null-terminated string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A null-terminated string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A null-terminated string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the *lpszTo* and *lpszCc* members, any addresses specified by this member will not be included in the header of the email message.

lpszDate

A null-terminated string that specifies the date and time for the message. This structure member may be NULL or point to an empty string. If the date is not specified, then the current date and time will be used by default. If a date is specified, it should be in the standard format as defined by RFC822.

lpszSubject

A null-terminated string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpszHeaders

A null-terminated string that specifies additional headers that should be included in the message. Header names should be separated from values by a colon, and multiple headers may be defined by separating them with a newline character. This structure member may be NULL, in which case no additional headers will be included in the message.

lpszText

A null-terminated string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the ***lpszHTML*** member must specify the body of the message.

lpszHTML

A null-terminated string which contains the message using HTML formatting. If the ***lpszText*** member is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the ***lpszText*** member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpszAttach

A null-terminated string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

An integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Value	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.
MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the

	application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Value	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using the **SmtpSubmitMessageEx** function. It is required that you specify a sender, at least one recipient and a message body. Other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the

structure.

Note that you must explicitly define the size of the structure by setting the value of the **dwSize** member variable. This ensures that the correct version of the structure is being passed to the function. This structure is not compatible with the **SmtpSubmitMessage** function and must only be used with **SmtpSubmitMessageEx**.

Email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* member cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

If you specify a message date by assigning a value to the *lpszDate* member, and it does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you provide input from a user, because in most cases they will not include the timezone and will assume the date and time they enter is for their current timezone.

If you wish to include additional headers in the message, you can specify them in a string. Each header consists of a name and value, separated by a colon (":") character. If you wish to define multiple headers, then you can separate them with a newline (e.g.: a linefeed character or combination of a carriage-return and linefeed). Extraneous leading and trailing whitespace are trimmed from header names and values. Invalid names or values will be ignored and will not generate an error.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SMTPSERVER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPSERVER

This structure provides connection information for a mail server and is used by the SmtpSubmitMessage function.

```
typedef struct _SMTPSERVER
{
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    UINT     nHostPort;
    UINT     nTimeout;
    DWORD    dwOptions;
    DWORD    dwReserved;
} SMTPSERVER, *LPSMTPSERVER;
```

Members

lpszHostName

A null-terminated string that specifies the host name or IP address of the mail server. This structure member cannot be NULL.

lpszUserName

A null-terminated string that specifies the username that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

lpszPassword

A null-terminated string that specifies the password that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

nHostPort

An integer value that specifies the port number used to establish the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit TLS connections, the default port number is 465.

nTimeout

An integer value that specifies the number of seconds that the client will wait for a response from the server before failing the operation. A value of zero specifies the default timeout period of 20 seconds.

dwOptions

An unsigned integer that specifies one or more options. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not

	specified, the library will not attempt to use any extended features. This option is automatically enabled if a username and password are specified, or if the connection is established on port 587, because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SMTPMESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMTPTRANSFERSTATUS

This structure is used by the SmtpGetTransferStatus function to return information about a message being submitted for delivery.

```
typedef struct _SMTPTRANSFERSTATUS
{
    DWORD  dwBytesTotal;
    DWORD  dwBytesCopied;
    DWORD  dwBytesPerSecond;
    DWORD  dwTimeElapsed;
    DWORD  dwTimeEstimated;
} SMTPTRANSFERSTATUS, *LPSMTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the size of the message cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol Library

Establish an interactive terminal session with a server.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSTNTV12.DLL
Version	12.0.3024.1286
LibID	552A583F-46C0-4A66-8BC5-B75B468763EE
Import Library	CSTNTV12.LIB
Dependencies	None
Standards	RFC 854

Overview

The Telnet protocol is used to establish a connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The class provides an interface for establishing the connection, negotiating certain options (such as whether characters will be echoed back to the client) and handling the standard I/O functions needed by the program.

The API also includes functions that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. This library can be combined with the Terminal Emulation API to provide terminal emulation services for a standard ANSI or DEC-VT220 terminal.

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the Secure Shell API.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1

(SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol Functions

Function	Description
TelnetAbort	Abort the current session and close the connection with the server
TelnetAsyncConnect	Connect asynchronously to the specified server
TelnetAttachThread	Attach the specified client handle to another thread
TelnetBreak	Send a break signal to the server
TelnetCancel	Cancel the current blocking operation
TelnetConnect	Connect to the specified server
TelnetCreateSecurityCredentials	Create a new security credentials structure
TelnetDeleteSecurityCredentials	Delete a previously created security credentials structure
TelnetDisableEvents	Disable asynchronous event notification
TelnetDisableTrace	Disable logging of network function calls
TelnetDisconnect	Disconnect from the current server
TelnetEnableEvents	Enable asynchronous event notification
TelnetEnableTrace	Enable logging of network function calls to a file
TelnetEventProc	Callback function that processes events generated by the client
TelnetFreezeEvents	Suspend asynchronous event processing
TelnetGetErrorString	Return a description for the specified error code
TelnetGetLastError	Return the last error code
TelnetGetMode	Return the current client mode
TelnetGetSecurityInformation	Return security information about the current client connection
TelnetGetStatus	Return the current client status
TelnetGetTerminalType	Return the current terminal type
TelnetGetTimeout	Return the number of seconds until an operation times out
TelnetInitialize	Initialize the library and validates the specified license key at runtime
TelnetIsBlocking	Determine if the client is blocked, waiting for information
TelnetIsConnected	Determine if the client is connected to the server
TelnetIsReadable	Determine if data can be read from the server
TelnetIsThere	Determine if the server is available
TelnetIsWritable	Determine if data can be written to the server
TelnetLogin	Login to the server using the specified username and password
TelnetRead	Read data returned by the server
TelnetReadLine	Read a line of text from the server and return it in a string buffer

TelnetRegisterEvent	Register an event callback function
TelnetSearch	Search for a specific character sequence in the data stream
TelnetSetLastError	Set the last error code
TelnetSetMode	Set the current client mode
TelnetSetTerminalType	Set the current terminal type
TelnetSetTimeout	Set the number of seconds until an operation times out
TelnetUninitialize	Terminate use of the library by the application
TelnetWrite	Write data to the server
TelnetWriteLine	Write a line of text to the server

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetAbort Function

```
INT WINAPI TelnetAbort(
   HCLIENT hClient
);
```

The **TelnetAbort** function aborts the current session and terminates the connection.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When the **TelnetAbort** function is called, the Telnet abort sequence is sent to the server and the connection to the server is terminated. Once this function returns, the client handle is no longer valid. If a program is currently executing on the server at the time this function is called, that program may be terminated as a result of the session being aborted. Applications should normally call **TelnetDisconnect** to gracefully disconnect from the server and should only use this function when the connection must be aborted immediately.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetBreak, TelnetCancel, TelnetIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetAsyncConnect Function

```
HCLIENT WINAPI TelnetAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **TelnetAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **TelnetConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TELNET_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
TELNET_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
TELNET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection

	will always be permitted. This option only affects connections using the TLS protocol.
TELNET_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
TELNET_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a standard connection with the server, then requests a secure connection during the option negotiation process.
TELNET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
TELNET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
TELNET_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the TELNET_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **TelnetGetLastError**.

Remarks

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the Secure Shell API.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **TelnetDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle.

The ownership of the handle may be transferred from one thread to another using the **TelnetAttachThread** function.

Specifying the TELNET_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetConnect, TelnetCreateSecurityCredentials, TelnetDeleteSecurityCredentials, TelnetDisconnect, TelnetInitialize, TelnetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetAttachThread Function

```
DWORD WINAPI TelnetAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **TelnetAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **TelnetAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **TelnetAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **TelnetCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **TelnetUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetConnect, TelnetDisconnect, TelnetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetBreak Function

```
INT WINAPI TelnetBreak(
   HCLIENT hClient
);
```

The **TelnetBreak** function sends a signal to the server which may terminate an application that is currently running. The actual response to the break signal depends on the application.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetAbort, TelnetCancel, TelnetRead, TelnetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetCancel Function

```
INT WINAPI TelnetCancel(
   HCLIENT hClient
);
```

The **TelnetCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When the **TelnetCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetAbort, TelnetBreak, TelnetIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetConnect Function

```
HCLIENT WINAPI TelnetConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **TelnetConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TELNET_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
TELNET_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
TELNET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using the TLS protocol.
TELNET_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
TELNET_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a

	standard connection with the server, then requests a secure connection during the option negotiation process.
TELNET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
TELNET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
TELNET_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This parameter is only used if the TELNET_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **TelnetGetLastError**.

Remarks

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the Secure Shell API.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **TelnetConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when

a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **TelnetAttachThread** function.

Specifying the TELNET_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetCreateSecurityCredentials, TelnetDeleteSecurityCredentials, TelnetDisconnect, TelnetInitialize, TelnetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetCreateSecurityCredentials Function

```
BOOL WINAPI TelnetCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **TelnetCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **TelnetDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **TelnetGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **TelnetConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
TelnetCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                                0,
                                NULL,
                                NULL,
                                lpszCertStore,
                                lpszCertName,
                                NULL,
                                &lpSecCred);

hClient = TelnetConnect(lpszHostName,
                        TELNET_PORT_SECURE,
                        TELNET_TIMEOUT,
                        TELNET_OPTION_SECURE,
                        lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetConnect, TelnetDeleteSecurityCredentials, TelnetGetSecurityInformation, SECURITYCREDENTIALS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetDeleteSecurityCredentials Function

```
VOID WINAPI TelnetDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **TelnetDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)
    TelnetDeleteSecurityCredentials(&lpSecCred);

TelnetUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetCreateSecurityCredentials, TelnetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetDisableEvents Function

```
INT WINAPI TelnetDisableEvents(
    HCLIENT hClient
);
```

The **TelnetDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetEnableEvents, TelnetFreezeEvents, TelnetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetDisableTrace Function

```
BOOL WINAPI TelnetDisableTrace();
```

The **TelnetDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetDisconnect Function

```
INT WINAPI TelnetDisconnect(
    HCLIENT hClient
);
```

The **TelnetDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetConnect, TelnetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetEnableEvents Function

```
INT WINAPI TelnetEnableEvents(
    HCLIENT hClient,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **TelnetEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **TelnetRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the ***hEventWnd*** parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetDisableEvents, TelnetFreezeEvents, TelnetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetEnableTrace Function

```
BOOL WINAPI TelnetEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **TelnetEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetEventProc Function

```
VOID CALLBACK TelnetEventProc(
    HCLIENT hClient,
    UINT nEvent,
    DWORD dwError,
    DWORD_PTR dwParam
);
```

The **TelnetEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **TelnetRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **TelnetRegisterEvent** function. The **TelnetEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetDisableEvents, TelnetEnableEvents, TelnetFreezeEvents, TelnetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetFreezeEvents Function

```
INT WINAPI TelnetFreezeEvents(
    HCLIENT hClient,
    BOOL bFreeze
);
```

The **TelnetFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetDisableEvents, TelnetEnableEvents, TelnetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetErrorString Function

```
INT WINAPI TelnetGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **TelnetGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **TelnetGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetGetLastError, TelnetSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetLastError Function

```
DWORD WINAPI TelnetGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **TelnetSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **TelnetGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **TelnetSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those functions which call **TelnetSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetErrorString, TelnetSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetMode Function

```
UINT WINAPI TelnetGetMode(
   HCLIENT hClient
);
```

The **TelnetGetMode** function returns the current client mode.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the current client mode. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The client mode is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters and single linefeeds will be automatically converted to carriage-return/linefeed sequences.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetSetMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetSecurityInformation Function

```
BOOL WINAPI TelnetGetSecurityInformation(
   HCLIENT hClient,
   LPSECURITYINFO lpSecurityInfo
);
```

The **TelnetGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

```
// Notifies the user if the connection is secure or not
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (TelnetGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
    }
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetConnect, TelnetDisconnect, SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetStatus Function

```
INT WINAPI TelnetGetStatus(
   HCLIENT hClient
);
```

The **TelnetGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
TELNET_STATUS_IDLE	The client is current idle and not sending or receiving data.
TELNET_STATUS_CONNECT	The client is establishing a connection with the server.
TELNET_STATUS_READ	The client is reading data from the server.
TELNET_STATUS_WRITE	The client is writing data to the server.
TELNET_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetIsBlocking, TelnetIsConnected, TelnetIsReadable, TelnetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetTerminalType Function

```
INT WINAPI TelnetGetTerminalType(
   HCLIENT hClient,
   LPCTSTR lpszTermType,
   INT nMaxLength
);
```

The **TelnetGetTerminalType** function returns the terminal type for the current client session.

Parameters

hClient

Handle to the client session.

lpszTermType

Points to a buffer which the current terminal type is copied into. This buffer should be at least 32 characters in length, including the terminating null character.

nMaxLength

Maximum number of characters that may be copied to the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the terminal type name. A value of zero indicates that no terminal type has been specified. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetSetTerminalType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetGetTimeout Function

```
INT WINAPI TelnetGetTimeout(
    HCLIENT hClient
);
```

The **TelnetGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetInitialize Function

```
BOOL WINAPI TelnetInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **TelnetInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **TelnetInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **TelnetUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **TelnetInitialize** or **TelnetUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetConnect, TelnetDisconnect, TelnetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetIsBlocking Function

```
BOOL WINAPI TelnetIsBlocking(
    HCLIENT hClient
);
```

The **TelnetIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **TelnetIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetCancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetIsConnected Function

```
BOOL WINAPI TelnetIsConnected(
    HCLIENT hClient
);
```

The **TelnetIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetIsBlocking, TelnetIsReadable, TelnetIsThere, TelnetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetIsReadable Function

```
BOOL WINAPI TelnetIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **TelnetIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetStatus, TelnetIsBlocking, TelnetIsConnected, TelnetIsThere, TelnetIsWritable, TelnetRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetIsThere Function

```
BOOL WINAPI TelnetIsThere(
   HCLIENT hClient
);
```

The **TelnetIsThere** function reports the response of the client to a "Are you there" command to the telnet server.

Parameters

hClient

Handle to the client session.

Return Value

The function returns a non-zero value if the server acknowledges a specific control sequence used to determine if a Telnet server is responsive. If the server does not respond, the function will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetStatus, TelnetIsBlocking, TelnetIsConnected, TelnetIsReadable, TelnetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetIsWritable Function

```
BOOL WINAPI TelnetIsWritable(
    HCLIENT hClient,
    INT nTimeout
);
```

The **TelnetIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetStatus, TelnetIsBlocking, TelnetIsConnected, TelnetIsReadable, TelnetIsThere, TelnetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetLogin Function

```
BOOL WINAPI TelnetLogin(
   HCLIENT hClient,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

The **TelnetLogin** function attempts to authenticate the user and log them in to the current session.

Parameters

hClient

Handle to the client session.

lpszUserName

A null-terminated string which specifies the name of the user to authenticate.

lpszPassword

A null-terminated string which specifies the password to be used when authenticating the user. If the user does not require a password, this parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

Caution: The Telnet protocol transmits all data, including login credentials, in plaintext and does not provide any inherent encryption. While this library supports optional TLS connections for Telnet, secure Telnet servers are rare in practice. This protocol is still supported for compatibility with legacy systems, but it is strongly recommended that you use secure protocols such as SSH whenever possible, especially when connecting over public or untrusted networks. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the Secure Shell API.

This function is used to authenticate a user, logging them into the server. This function is specifically designed to work with most UNIX based servers, and may work with other servers that use a similar login process. The function works by scanning the data stream for a username prompt and then replying with the specified username. If that is successful, it will then scan for a password prompt and provide the specified password. If no recognized prompt is found, or if the server responds with an error indicating that the username or password is invalid, the function will fail.

If the **TelnetLogin** function succeeds, the next call to **TelnetRead** by the client will return any welcome message to the user. This is typically followed by a command prompt where the user can enter commands to be executed on the server. The data sent by the server during the login process is discarded and not available when the function returns. If the client requires this information, use the **TelnetSearch** function to automate the login process instead.

Because the **TelnetLogin** function is designed for UNIX based systems, it may not work with servers running on other operating system platforms such as Windows or VMS. In this case, applications should use the **TelnetSearch** function to search for the appropriate login prompts in the data stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetConnect, TelnetIsConnected, TelnetIsReadable, TelnetRead, TelnetSearch

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetRead Function

```
INT WINAPI TelnetRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **TelnetRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When **TelnetRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetIsReadable, TelnetSearch, TelnetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetReadLine Function

```
BOOL WINAPI TelnetReadLine(
   HCLIENT hClient,
   LPTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **TelnetReadLine** function reads up to a line of data and returns it in a string buffer.

Parameters

hClient

Handle to the client session.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetReadLine** function reads data sent by the server and copies it into a specified string buffer. Unlike the **TelnetRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **TelnetReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the server. The data is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, read the data and then restore the client to asynchronous operation. If another client operation is attempted while **TelnetReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) client connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **TelnetRead** and **TelnetReadLine** function calls can be intermixed, however be aware that **TelnetRead** will consume any data that has already been buffered by the **TelnetReadLine** function and this may have unexpected results.

Unlike the **TelnetRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = TelnetReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = TelnetGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetIsReadble, TelnetRead, TelnetWrite, TelnetWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetRegisterEvent Function

```
INT WINAPI TelnetRegisterEvent(
   HCLIENT hClient,
   UINT nEvent,
   TELNETEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **TelnetRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
TELNET_EVENT_CONNECT	The connection to the server has completed.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **TelnetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86

(32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetRegisterEvent** function associates a callback function with a specific event. The event handler is an **TelnetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetDisableEvents, TelnetEnableEvents, TelnetEventProc, TelnetFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetSearch Function

```
BOOL WINAPI TelnetSearch(
   HCLIENT hClient,
   LPCTSTR lpszString,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwReserved
);
```

The **TelnetSearch** function searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

hClient

Handle to the client session.

lpszString

A null-terminated string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpvBuffer

A buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the function returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the ***lpvBuffer*** parameter is NULL, this parameter should also be NULL.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds and the character sequences was found in the data stream, the return value is non-zero. If the function fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetSearch** function searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as logging in a user and executing a command. The function collects the output from the server and stores it in the buffer specified by the ***lpvBuffer*** parameter. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

The ***lpvBuffer*** parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated, the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
LPCTSTR lpszUserName = "abc123\r\n";
LPCTSTR lpszPassword = "secret\r\n";
LPCTSTR lpszCommand = "/bin/ls -l\r\n";
HGLOBAL hgblOutput = NULL;
DWORD cbOutput = 0;
BOOL bResult;

// Search for the login prompt issued by the server

bResult = TelnetSearch(hClient,
                       _T("ogin: "),
                       NULL,
                       NULL,
                       0);

// If the Login: prompt was found, then write out the
// username and search for the Password: prompt; note
// that the username, password and command strings are
// terminated with a carriage-return/linefeed sequence
// which the server will see as the user pressing the
// Enter or Return key on the keyboard

if (bResult)
{
    TelnetWrite(hClient,
                (LPBYTE)lpszUserName,
                lstrlen(lpszUserName));

    bResult = TelnetSearch(hClient,
                           _T("word: "),
                           NULL,
                           NULL,
                           0);
}

// If the Password: prompt was found, write out the
// password and then search for the shell prompt;
// the prompt may be different, depending on what
// operating system and shell is being used

if (bResult)
{
    TelnetWrite(hClient,
                (LPBYTE)lpszPassword,
                lstrlen(lpszPassword));

    bResult = TelnetSearch(hClient,
                           _T("$ "),
                           NULL,
```

```
                            NULL,
                            0);
}

// If the shell prompt was found, issue the command
// and capture the output into the hgblBuffer global
// memory buffer; the cbBuffer variable will contain
// the actual number of bytes in the buffer when the
// function returns

if (bResult)
{
    TelnetWrite(hClient,
                (LPBYTE)lpszCommand,
                lstrlen(lpszCommand));

    bResult = TelnetSearch(hClient,
                           _T("$ "),
                           &hgblOutput,
                           &cbOutput,
                           0);
}

// Write the contents of the output buffer to the
// standard output stream

if (bResult)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    if (lpBuffer)
        fwrite(lpBuffer, 1, cbBuffer, stdout);

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetIsBlocking, TelnetIsReadable, TelnetLogin, TelnetRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetSetLastError Function

```
VOID WINAPI TelnetSetLastError(
    DWORD dwErrorCode
);
```

The **TelnetSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those functions which call **TelnetSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **TelnetGetLastError** function. The use of **TelnetGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetErrorString, TelnetGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetSetMode Function

```
UINT WINAPI TelnetSetMode(
   HCLIENT hClient,
   UINT nMode,
   BOOL bEnable
);
```

The **TelnetSetMode** function sets one or more client modes for the specified session.

Parameters

hClient

Handle to the client session.

nMode

The client mode. This value is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters, and single linefeed characters will be converted to carriage-return/linefeed sequences.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the function succeeds, the return value is the previous mode. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetSetTerminalType Function

```
INT WINAPI TelnetSetTerminalType(
   HCLIENT hClient,
   LPCTSTR lpszTermType
);
```

The **TelnetSetTerminalType** function sets the terminal type for the current client session.

Parameters

hClient

Handle to the client session.

lpszTermType

Points to a string which specifies the terminal type.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetGetTerminalType

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetSetTimeout Function

```
INT WINAPI TelnetSetTimeout(
   HCLIENT hClient,
   INT nTimeout
);
```

The **TelnetSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetUninitialize Function

```
VOID WINAPI TelnetUninitialize();
```

The **TelnetUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **TelnetInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **TelnetUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **TelnetUninitialize** for every successful call to **TelnetInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetDisconnect, TelnetInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetWrite Function

```
INT WINAPI TelnetWrite(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **TelnetWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the TELNET_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib

See Also

TelnetRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

TelnetWriteLine Function

```
BOOL WINAPI TelnetWriteLine(
   HCLIENT hClient,
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **TelnetWriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

hClient

Handle to the client session.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the server. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the server.

lpnLength

A pointer to an integer value which will contain the number of characters written to the server, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetWriteLine** function writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **TelnetWrite** function which writes arbitrary bytes of data to the server, this function is specifically designed to write a single line of text data from a string.

If the ***lpszBuffer*** string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the ***lpnLength*** parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **TelnetWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support sending raw Unicode data to the server. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, send the data and then

restore the client to asynchronous operation. If another network operation is attempted while **TelnetWriteLine** is blocked sending data to the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **TelnetWrite** and **TelnetWriteLine** function calls can be safely intermixed.

Unlike the **TelnetWrite** function, it is possible for data to have been written to the server if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstntv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

TelnetIsWritable, TelnetRead, TelnetReadLine, TelnetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Telnet Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation Library

Emulate an ANSI or DEC VT-220 character mode display terminal.

Reference

- Functions
- Data Structures
- Control Sequences
- Error Codes

Library Information

File Name	CSNVTV12.DLL
Version	12.0.3024.1286
LibID	9A302CB9-6BEE-4D11-9785-76E7624BD130
Import Library	CSNVTV12.LIB
Dependencies	None

Overview

The Terminal Emulation library provides a comprehensive API for emulating an ANSI or DEC-VT220 terminal, with full support for all standard escape and control sequences, color mapping and other advanced features. The library functions provide both a high level interface for parsing escape sequences and updating a display, as well as lower level primitives for directly managing the virtual display, such as controlling the individual display cells, moving the cursor position and specifying display attributes.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or

later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation Functions

Function	Description
NvtClearDisplay	Clear the specified display
NvtConvertDisplayPos	Convert between pixel and screen coordinates
NvtCopySelectedText	Copy the selected text to the clipboard
NvtCreateDisplay	Create a new virtual terminal emulation display
NvtDeleteChar	Delete the specified number of characters
NvtDeleteLine	Delete the current line, shifting remaining lines up
NvtDestroyDisplay	Destroy the specified virtual display
NvtEraseChar	Erase the specified number of characters
NvtEraseLine	Erase the current line
NvtGetCursorPos	Return the current cursor position
NvtGetDisplayAttributes	Return the current display attributes
NvtGetDisplayCell	Return information about the specified cell in the virtual display
NvtGetDisplayCellSize	Return the size of the display character cells
NvtGetDisplayColor	Get the current virtual display colors
NvtGetDisplayColorMap	Return the virtual display color table
NvtGetDisplayDC	Get the current virtual display device context
NvtGetDisplayEmulation	Get the current terminal emulation type
NvtGetDisplayFont	Get the current virtual display font
NvtGetDisplayInfo	Return information about the virtual display
NvtGetDisplayLine	Return a line of text from the virtual display
NvtGetDisplayMode	Get the current virtual display mode
NvtGetDisplayRect	Return the rectangle for the display window client area
NvtGetDisplayScrollPos	Get the display scroll box position
NvtGetDisplaySize	Return the current size of the virtual display
NvtGetDisplayText	Get the specified block of text from the virtual display
NvtGetDisplayWindow	Get the current virtual display window
NvtGetMappedKey	Return the escape sequence for the mapped key
NvtGetScrollRegion	Return the current scrolling region
NvtGetSelectedText	Return the currently selected text
NvtGetTextColor	Return the current text foreground or background color
NvtInitialize	Initialize the library for use by the client

NvtInsertChar	Insert the specified number of characters
NvtInsertLine	Insert a line, shifting the remaining lines down
NvtRefreshDisplay	Refresh the specified display
NvtResetDisplay	Reset the virtual display
NvtResetMappedKeys	Reset the mapped key table to default values
NvtResizeDisplay	Resize the virtual display
NvtRestoreCursor	Restore the saved cursor position and text attributes
NvtSaveCursor	Save the current cursor position and text attributes
NvtScrollDisplay	Scroll the virtual display
NvtSelectDisplayText	Select a region of the virtual display
NvtSetCursorPos	Set the current cursor position
NvtSetDisplayAttributes	Set the current display attributes
NvtSetDisplayBackColor	Set the background color for the virtual display
NvtSetDisplayBoldColor	Set the bold color for the virtual display
NvtSetDisplayCell	Set the value of a character cell in the virtual display
NvtSetDisplayColor	Set the virtual display colors
NvtSetDisplayColorMap	Set the virtual display color table
NvtSetDisplayDC	Set the current virtual display device context
NvtSetDisplayEmulation	Set the current terminal emulation type
NvtSetDisplayFocus	Set the focus on the virtual display
NvtSetDisplayFont	Set the current virtual display font
NvtSetDisplayFontName	Set the current virtual display font by name
NvtSetDisplayForeColor	Set the foreground color for the virtual display
NvtSetDisplayMode	Set the current virtual display mode
NvtSetDisplayScrollPos	Set the display scroll box position
NvtSetDisplaySize	Set the size of the virtual display
NvtSetDisplayWindow	Set the current virtual display window
NvtSetMappedKey	Set the escape sequence for the specified key
NvtSetScrollRegion	Set the current scrolling region
NvtSetTextColor	Set the current text foreground or background color
NvtTranslateMappedKey	Translate the keypress to a mapped key escape sequence
NvtUninitialize	Terminate use of the library by the application
NvtUpdateCaret	Update the display window caret
NvtUpdateDisplay	Update the window attached to the virtual display
NvtWriteDisplay	Write the specified buffer to the virtual display

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtClearDisplay Function

```
BOOL WINAPI NvtClearDisplay(
   HDISPLAY hDisplay,
   UINT nMode
);
```

The **NvtClearDisplay** function clears the specified display, erasing the text and clearing any attributes.

Parameters

hDisplay

Handle to the virtual display.

nMode

Mode which specifies how the display will be cleared. The following values may be used:

Value	Description
NVT_CLEAR_EOS	The display is cleared from the current cursor position to the end of the display. The cursor position is not changed.
NVT_CLEAR_TOS	The display is cleared from the beginning of the display to the current cursor position. The cursor position is not changed.
NVT_CLEAR_ALL	The entire display is cleared and the cursor is repositioned to the upper left corner of the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtDeleteChar, NvtDeleteLine, NvtEraseChar, NvtEraseLine, NvtResetDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtConvertDisplayPos Function

```
BOOL WINAPI NvtConvertDisplayPos(
    HDISPLAY hDisplay,
    INT nMethod,
    INT xPos,
    INT yPos,
    LPPOINT lppt
);
```

The **NvtConvertDisplayPos** function converts the specified X,Y position and stores the result in the POINT structure provided by the caller.

Parameters

hDisplay

Handle to the virtual display.

nMethod

An integer value which specifies the conversion method to use. This may be one of the following values:

Value	Description
NVT_CURSOR_TO_PIXELS	Convert cursor X,Y coordinates to the current window X,Y pixel coordinates. An error is returned if the coordinates are out of bounds for the current display.
NVT_PIXELS_TO_CURSOR	Convert window X,Y pixel coordinates to cursor X,Y coordinates. If the point is outside of the bounds of the display, it is normalized to account for mouse capture.

xPos

An integer value which specifies the X position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the ***nMethod*** parameter.

yPos

An integer value which specifies the Y position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the ***nMethod*** parameter.

lppt

A pointer to a POINT structure which will contain the converted coordinates.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetCursorPos, NvtGetDisplayCellSize, NvtGetDisplayScrollPos, NvtGetDisplaySize,

NvtSetCursorPos, NvtSetDisplayScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtCopySelectedText Function

```
BOOL WINAPI NvtCopySelectedText(
    HDISPLAY hDisplay
);
```

The **NvtCopySelectedText** function copies any selected text to the system clipboard.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetSelectedText, NvtSelectDisplayText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtCreateDisplay Function

```
HDISPLAY WINAPI NvtCreateDisplay(
    HWND hWnd,
    HFONT hFont,
    UINT nEmulation,
    UINT nColumns,
    UINT nRows
);
```

The **NvtCreateDisplay** function creates a new virtual terminal emulation display using the specified window and font.

Parameters

hWnd

Handle to the window that will be used to display the virtual terminal.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process escape sequences.
NVT_EMULATION_ANSI	The virtual display processes ANSI escape sequences for screen management and cursor positioning. This emulation also supports escape sequences to control the foreground and background color. The default keymap for ANSI function key escape sequences will be selected.
NVT_EMULATION_VT100	The virtual display processes DEC VT-100 escape sequences for screen management and cursor positioning. The default keymap for a DEC VT-100 terminal will be selected.
NVT_EMULATION_VT220	The virtual display processes DEC VT-220 escape sequences for screen management and cursor positioning. This emulation also supports DEC VT-320 escape sequences to control the foreground and background color. The default keymap for a DEC VT-220 terminal will be selected.

nColumns

The maximum number of columns used by the virtual display. This value must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. This value must be at least 5, and no greater than 127.

Return Value

If the function succeeds, the return value is a handle to the virtual display. If the function fails, the return value is INVALID_DISPLAY.

Remarks

The default colors for the display is a black background and white foreground. If ANSI terminal emulation is selected, bold characters will be displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtDestroyDisplay, NvtGetDisplayInfo, NvtInitialize, NvtUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtDeleteChar Function

```
BOOL WINAPI NvtDeleteChar(
   HDISPLAY hDisplay,
   INT nChars
);
```

The **NvtDeleteChar** function deletes the specified number of characters from the display, shifting the characters that follow to the left. The characters are deleted from the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

nChars

Number of characters to delete from the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To erase characters from the display without affecting the characters that follow, use the **NvtEraseChar** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtClearDisplay, NvtDeleteLine, NvtEraseChar, NvtEraseLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtDeleteLine Function

```
BOOL WINAPI NvtDeleteLine(
   HDISPLAY hDisplay
);
```

The **NvtDeleteLine** function deletes the current line, shifting up the lines that follow.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To erase a line from the display without affecting the lines that follow, use the **NvtEraseLine** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtClearDisplay, NvtDeleteChar, NvtEraseChar, NvtEraseLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtDestroyDisplay Function

```
VOID WINAPI NvtDestroyDisplay(
   HDISPLAY hDisplay
);
```

The **NvtDestroyDisplay** function releases the memory allocated for the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

None.

Remarks

If the display font was specified when the display was created, then it is the responsibility of the application to delete the font object using the **DeleteObject** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtInitialize, NvtUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtEraseChar Function

```
BOOL WINAPI NvtEraseChar(
   HDISPLAY hDisplay,
   INT nChars
);
```

The **NvtEraseChar** function erases the specified number of characters from the display, without affecting the position of the characters that follow. The characters are erased from the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

nChars

Number of characters to erase from the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To delete characters from the display and shift the remaining characters to the left, use the **NvtDeleteChar** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtClearDisplay, NvtDeleteChar, NvtDeleteLine, NvtEraseLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtEraseLine Function

```
BOOL WINAPI NvtEraseLine(
   HDISPLAY hDisplay,
   UINT nMode
);
```

The **NvtEraseLine** function erases the current line without affecting the lines that follow.

Parameters

hDisplay

Handle to the virtual display.

nMode

Mode which specifies how the line will be erased. The following values may be used:

Value	Description
The line is erased from the current cursor position to the end of the line.	
The line is cleared from the beginning of the line to the current cursor position.	
The entire line is cleared.	

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To delete a line from the display and shift the remaining lines up, use the **NvtDeleteLine** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtClearDisplay, NvtDeleteChar, NvtDeleteLine, NvtEraseChar

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetCursorPos Function

```
BOOL WINAPI NvtGetCursorPos(
   HDISPLAY hDisplay,
   LPINT lpnCursorX,
   LPINT lpnCursorY
);
```

The **NvtGetCursorPos** function returns the current cursor column and row position on the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpnCursorX

Address of the variable that will be set to the column of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

lpnCursorY

Address of the variable that will be set to the row of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

Return Value

If the function succeeds, the return value is a non-zero value. If the handle to the virtual display is invalid, the function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtConvertDisplayPos, NvtSetCursorPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayAttributes Function

```
UINT WINAPI NvtGetDisplayAttributes(
    HDISPLAY hDisplay
);
```

The **NvtGetDisplayAttributes** function returns the current display attributes which have been set, either explicitly by the client or as the result of an escape sequence parsed by the emulator.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value the current display attributes. If the function fails, the return value is NVT_ERROR.

The following table lists the valid attributes:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayCell, NvtSetDisplayAttributes

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayCell Function

```
DWORD WINAPI NvtGetDisplayCell(
    HDISPLAY hDisplay,
    INT xPos,
    INT yPos
);
```

The **NvtGetDisplayCell** function returns information about the specified character cell in the display.

Parameters

hDisplay

Handle to the virtual display.

xPos

An integer value which specifies the X cursor position in the display.

yPos

An integer value which specifies the Y cursor position in the display.

Return Value

If the function succeeds, the return value is the cell character and attributes. If the function fails, it returns NVT_ERROR.

Remarks

The value returned by the **NvtGetDisplayCell** function is an unsigned 32-bit integer value, where the low order word specifies the ANSI character stored at that position and the high order word specifies the display attributes for that cell.

The character cell attributes may be one or more of the following values:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayAttributes, NvtGetDisplayCellSize, NvtSetDisplayAttributes

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayCellSize Function

```
BOOL WINAPI NvtGetDisplayCellSize(
    HDISPLAY hDisplay,
    LPSIZE lpCellSize
);
```

The **NvtGetDisplayCellSize** function returns the size of a character cell in pixels.

Parameters

hDisplay

Handle to the virtual display.

lpCellSize

A pointer to a SIZE structure which will contain the size of a character cell in pixels.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayCellSize** function is used to determine the size of a character cell in the display. This can be useful when the application needs to determine where a display cell is physically located within the virtual display window.

To convert between display and window coordinates, use the **NvtConvertDisplayPos** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtConvertDisplayPos, NvtGetDisplayCell

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayColor Function

```
COLORREF WINAPI NvtGetDisplayColor(
    HDISPLAY hDisplay,
    UINT nColorIndex,
    BOOL bForeground
);
```

The **NvtGetDisplayColor** function returns the color used by the virtual display to display text with a specific attribute.

Parameters

hDisplay

Handle to the virtual display.

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Value	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the function succeeds, the return value is the RGB value of the specified color. If the function fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtSetDisplayBackColor, NvtSetDisplayBoldColor, NvtSetDisplayColor, NvtSetDisplayForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayColorMap Function

```
BOOL WINAPI NvtGetDisplayColorMap(
    HDISPLAY hDisplay,
    COLORREF* lpColor,
    INT nMaxColors
);
```

The **NvtGetDisplayColorMap** function returns the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

hDisplay

Handle to the virtual display.

lpColor

A pointer to an array of COLORREF values which will contain the color values currently being used in the virtual display.

nMaxColors

The maximum number of colors which may be stored in the color array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid, the pointer to the color table is NULL or the maximum number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **NvtGetDisplayColorMap** function is useful for determining what values are being used when a color attribute is set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
Black	RGB(0,0,0)			
Red	000000A0h	RGB(160,0,0)		
Green	0000A000h	RGB(0,160,0)		
Yellow	0000A0A0h	RGB(160,160,0)		
Blue	00A00000h	RGB(0,0,160)		
Magenta	00A000A0h	RGB(160,0,160)		
Cyan	00A0A000h	RGB(0,160,160)		
White	00E0E0E0h	RGB(224,224,224)		
Gray	00C0C0C0h	RGB(192,192,192)		
Light Red	008080FFh	RGB(255,128,128)		

Light Green	0090EE90h	RGB(144,238,144)
Light Yellow	00C0FFFFh	RGB(255,255,192)
Light Blue	00E6D8ADh	RGB(173,216,230)
Light Magenta	00FFC0FFh	RGB(255,192,255)
Light Cyan	00FFFFE0h	RGB(224,255,255)
High White	00FFFFFFh	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

NvtGetDisplayColorMap is typically used in conjunction with the **NvtSetDisplayColorMap** function to load the current color values and then make selective changes to the actual RGB color value that is used when a color attribute is set. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if ( NvtGetDisplayColorMap(hDisplay, rgbColor, 16) )
{
     rgbColor[7] = rgbColor[15];
     NvtSetDisplayColorMap(hDisplay, rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColor, NvtGetTextColor, NvtSetDisplayColor, NvtSetDisplayColorMap, NvtSetTextColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayDC Function

```
HDC WINAPI NvtGetDisplayDC(
   HDISPLAY hDisplay
);
```

The **NvtGetDisplayDC** returns the device context that has been specified for the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is a handle to the device context. If the function fails, or no device context has been specified, the return value is NULL.

Remarks

Normally there is no device context explicitly set for the display. Instead, the device context is dynamically created and released when the virtual display is updated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtSetDisplayDC, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayEmulation Function

```
UINT WINAPI NvtGetDisplayEmulation(
    HDISPLAY hDisplay
);
```

The **NvtGetDisplayEmulation** function returns the current virtual terminal emulation type.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is the terminal emulation type and may contain one of the following values:

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

If the function fails, because an invalid display handle was specified, the return value will be NVT_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtResetDisplay, NvtSetDisplayEmulation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayFont Function

```
HFONT WINAPI NvtGetDisplayFont(
    HDISPLAY hDisplay
);
```

The **NvtGetDisplayFont** function returns the current font handle being used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is a handle to the font. If the function fails, it returns NULL.

Remarks

The handle returned by this function should not be released, and the font object should never be directly modified by the application. Functions that return information about the font, such as **GetTextMetrics**, may be safely called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtSetDisplayFont, NvtSetDisplayFontName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayInfo Function

```
BOOL WINAPI NvtGetDisplayInfo(
   HDISPLAY hDisplay,
   LPNVTDISPLAYINFO lpvdi
);
```

The **NvtGetDisplayInfo** function returns information about the specified virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpvdi

Pointer to a NVTDISPLAYINFO structure which contains information about the virtual display, including the display window, font, cursor and scrolling position.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtGetCursorPos, NvtGetDisplayFont, NvtGetDisplayMode, NvtGetDisplayRect, NvtGetDisplayScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayLine Function

```
INT WINAPI NvtGetDisplayLine(
   HDISPLAY hDisplay,
   INT nRow,
   LPTSTR lpszText,
   INT nMaxLength
);
```

The **NvtGetDisplayLine** function copies a block of text from the specified virtual display into a string buffer.

Parameters

hDisplay

Handle to the virtual display.

nRow

The row in the virtual display to return the line of text from. The first row in the display is zero. If the value -1 is specified, the row where the cursor is currently located will be used.

lpszText

Pointer to the buffer that the display text will be copied to, terminated with a null character character.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayLine** function allows the application to copy the contents of the display at a specific row. Note that the buffer must be large enough to accommodate the text and the null character terminator. Unlike the **NvtGetDisplayText** function, any trailing whitespace in the specified row is not copied to the buffer.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtGetDisplayText, NvtUpdateDisplay, NvtWriteDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayMode Function

```
UINT WINAPI NvtGetDisplayMode(
    HDISPLAY hDisplay
);
```

The **NvtGetDisplayMode** function returns the current virtual display modes.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is the display mode for the virtual display. If the function fails, it returns zero.

Remarks

The display mode is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other functions. The following values are recognized.

Value	Description
NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.

NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT (horizontal tab) control sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less then that total number of columns in the virtual display. Disabling this mode prevents a horizontal scrollbar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less then that total number of rows in the virtual display. Disabling this mode prevents a vertical scrollbar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayEmulation, NvtGetDisplayInfo, NvtSetDisplayMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayRect Function

```
BOOL WINAPI NvtGetDisplayRect(
   HDISPLAY hDisplay,
   LPRECT lpRect
);
```

The **NvtGetDisplayRect** returns the client rectangle for the virtual display window.

Parameters

hDisplay

Handle to the virtual display.

lpRect

Pointer to a RECT structure which will contain the client rectangle values when the function returns.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayInfo, NvtGetDisplaySize, NvtResizeDisplay, NvtSetDisplaySize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayScrollPos Function

```
INT WINAPI NvtGetDisplayScrollPos(
    HDISPLAY hDisplay,
    INT nScrollBar
);
```

The **NvtGetDisplayScrollPos** function gets the position of the scroll box for the specified scroll bar.

Parameters

hDisplay

Handle to the virtual display.

nScrollBar

Specifies the scroll bar to return the position for. This parameter can be one of the following values:

Value	Description
SB_HORZ	Gets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Gets the position of the scroll box in the display's standard vertical scroll bar.

Return Value

If the function succeeds, the return value is the position of the scroll box. If the function fails, it returns -1.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtSetDisplayScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplaySize Function

```
BOOL WINAPI NvtGetDisplaySize(
   HDISPLAY hDisplay,
   LPSIZE lpSize
);
```

The **NvtGetDisplaySize** returns the size of the virtual display in columns and rows.

Parameters

hDisplay

Handle to the virtual display.

lpSize

Pointer to a SIZE structure which will contain the size of the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The **NvtGetDisplaySize** function returns the number of columns and rows in the virtual display. To convert this to pixels, use the **NvtGetDisplayCellSize** function to determine the size of a character cell and multiply that by the number of columns and/or rows.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayCellSize, NvtGetDisplayInfo, NvtGetDisplayRect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayText Function

```
INT WINAPI NvtGetDisplayText(
   HDISPLAY hDisplay,
   INT nOffset,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **NvtGetDisplayText** function copies a block of text from the specified virtual display into a string buffer.

Parameters

hDisplay

Handle to the virtual display.

nOffset

Offset into the virtual display buffer. A value of -1 specifies that the current cursor position should be used as the offset.

lpszBuffer

Pointer to the buffer that the display text will be copied to, terminated with a null character character.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayText** function allows the application to copy the contents of the display at a specific location. Note that the buffer must be large enough to accommodate the text and the null character terminator. To copy an entire row of text in the display, use the **NvtGetDisplayLine** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtGetDisplayLine, NvtUpdateDisplay, NvtWriteDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayWindow Function

```
HWND WINAPI NvtGetDisplayWindow(
    HDISPLAY hDisplay
);
```

The **NvtGetDisplayWindow** returns the handle to the window being used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is a handle to the virtual display window. If the function fails, it returns NULL.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetMappedKey Function

```
BOOL WINAPI NvtGetMappedKey(
   HDISPLAY hDisplay,
   UINT nMappedKey,
   LPTSTR lpszKeyBuffer,
   UINT cchKeyBuffer
);
```

The **NvtGetMappedKey** function returns the escape sequence mapped to the specified key.

Parameters

hDisplay

Handle to the virtual display.

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (function) keys for the current emulation.

lpszKeyBuffer

Address of the buffer which will receive the escape sequence mapped to the specified key.

cchKeyBuffer

The maximum number of characters that may be copied into the key buffer string, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero and the escape sequence for the mapped key is copied into the specified buffer. If the key has not been mapped, and there is no default escape sequence defined, then the function will return zero.

Remarks

The **NvtGetMappedKey** function returns the escape sequence that has been mapped to a special key. This function can be used to determine what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 function key). If a sequence has not been explicitly mapped through a call to **NvtSetMappedKey**, the default sequence for the current emulation will be returned.

Note that the current display mode, such as whether or not the emulator is in application mode or not, should be considered when determining which mapped key to use. For example, if the emulator is not in application mode and the user presses the up-arrow key, the sequence mapped to the NVT_UP key should be sent to the server. However, if the emulator is in application mode, the sequence mapped to the NVT_APPUP key should be sent instead. This is automatically handled by the **NvtTranslateMappedKey** function, so it is recommended that it be used when mapping a virtual keypress to the appropriate escape sequence.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtGetDisplayMode, NvtResetMappedKeys, NvtSetDisplayMode, NvtSetMappedKey, NvtTranslateMappedKey

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetScrollRegion Function

```
BOOL WINAPI NvtGetScrollRegion(
    HDISPLAY hDisplay,
    LPINT lpnTop,
    LPINT lpnBottom
);
```

The **NvtGetScrollRegion** function returns the top and bottom rows of the current scrolling region.

Parameters

hDisplay

Handle to the virtual display.

lpnTop

Address of the variable that will be set to the top row of the current scrolling region. If no scrolling region has been defined, the value will be 0, the first row in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

lpnBottom

Address of the variable that will be set to the bottom row of the current scrolling region. If no scrolling region has been defined, the value will be one less than the maximum number of rows in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

Return Value

If the function succeeds, it will return zero. If the handle to the display is invalid, the function will return zero.

Remarks

The **NvtGetScrollRegion** function allows an application to determine the top and bottom rows of the current scrolling region. By default, the scrolling region is the entire virtual display, however this may be changed through a call to **NvtSetScrollRegion** or an ANSI escape sequence. If the display is in origin mode, note that the cursor cannot be positioned outside of the current scrolling region.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayMode, NvtSetDisplayMode, NvtSetScrollRegion

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetSelectedText Function

```
INT WINAPI NvtGetSelectedText(
   HDISPLAY hDisplay,
   LPTSTR lpszBuffer,
   INT nMaxLength
);
```

The **NvtGetSelectedText** function copies the currently selected text into the specified buffer.

Parameters

hDisplay

Handle to the virtual display.

lpszBuffer

Pointer to the buffer that the selected text will be copied to, terminated with a null character character.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtCopySelectedText, NvtSelectDisplayText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtGetTextColor Function

```
BOOL WINAPI NvtGetTextColor(
   HDISPLAY hDisplay,
   COLORREF * lprgbColor,
   BOOL bForeground
);
```

The **NvtGetTextColor** function returns the current foreground or background text color.

Parameters

hDisplay

Handle to the virtual display.

lprgbColor

A pointer to a COLORREF variable which will contain the current foreground or background text color.

bForeground

A boolean value which determines if the foreground or background color is returned. A value of TRUE indicates that the foreground color should be returned, otherwise the background color is returned.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. Failure indicates that the handle to the display is invalid, the display does not support text color attributes, or the pointer to the color value is NULL.

Remarks

This function is used to return the current foreground or background color, as determined by the text attribute. The RGB color value for a color attribute is determined by the virtual display's color table.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColorMap, NvtSetDisplayColorMap, NvtSetTextColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtInitialize Function

```
BOOL WINAPI NvtInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **NvtInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NvtGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **NvtInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **NvtUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **NvtInitialize** or **NvtUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtCreateDisplay, NvtDestroyDisplay, NvtResetDisplay, NvtUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtInsertChar Function

```
BOOL WINAPI NvtInsertChar(
   HDISPLAY hDisplay,
   INT nChars
);
```

The **NvtInsertChar** function inserts one or more space characters at the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

nChars

Number of space characters to insert at the current cursor position.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the cursor position. Inserting space characters may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtDeleteChar, NvtEraseChar, NvtInsertLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtInsertLine Function

```
BOOL WINAPI NvtInsertLine(
   HDISPLAY hDisplay
);
```

The **NvtInsertLine** function inserts an empty line at the current row, shifting the remaining lines down.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the cursor position. Inserting a line may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtDeleteLine, NvtEraseLine, NvtInsertChar

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtRefreshDisplay Function

```
BOOL WINAPI NvtRefreshDisplay(
   HDISPLAY hDisplay,
   BOOL bUpdate
);
```

The **NvtRefreshDisplay** function refreshes the specified virtual display, updating the current scroll position and caret.

Parameters

hDisplay

Handle to the virtual display.

bUpdate

Boolean flags which specifies if the display window is to be updated. If set, the window client area is invalidated and the virtual display is redrawn.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtUpdateCaret, NvtUpdateDisplay, NvtWriteDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtResetDisplay Function

```
BOOL WINAPI NvtResetDisplay(
   HDISPLAY hDisplay,
   HWND hWnd,
   HFONT hFont,
   UINT nColumns,
   UINT nRows
);
```

The **NvtResetDisplay** resets the virtual terminal display, using the new window, font, columns and rows. This function should be used when the virtual display must be attached to a different window, or the number of rows or columns must be changed.

Parameters

hDisplay

Handle to the virtual display.

hWnd

Handle to the window that will be used to display the virtual terminal. This parameter may be NULL, in which case the current window will be used.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case the current font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nColumns

The maximum number of columns used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 127.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtClearDisplay, NvtCreateDisplay, NvtRefreshDisplay, NvtResizeDisplay, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtResetMappedKeys Function

```
BOOL WINAPI NvtResetMappedKeys(
    HDISPLAY hDisplay
);
```

The **NvtResetMappedKeys** function resets all mapped function keys to their default values, based on the current emulation.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the handle to the display is invalid, the function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetMappedKey, NvtSetMappedKey, NvtTranslateMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtResizeDisplay Function

```
BOOL WINAPI NvtResizeDisplay(
   HDISPLAY hDisplay,
   INT cxClient,
   INT cyClient
);
```

The **NvtResizeDisplay** function resizes the virtual display to the specified width and height.

Parameters

hDisplay

Handle to the virtual display.

cxClient

New width of the display window in pixels. If this value is zero, the width of the virtual display remains unchanged.

cyClient

New height of the display window in pixels. If this value is zero, the height of the virtual display remains unchanged.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function resizes the virtual display and updates the scrolling information. Typically this function is called when the display window receives a WM_SIZE message, causing the virtual display to match the size of the window's client area.

This function will not change the size of the display window. To change the size of the display window, use the **SetWindowPos** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtGetDisplayRect, NvtRefreshDisplay, NvtResetDisplay, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtRestoreCursor Function

```
BOOL WINAPI NvtRestoreCursor(
   HDISPLAY hDisplay
);
```

The **NvtRestoreCursor** function restores the cursor position and text attributes to their previous values.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

Use the **NvtSaveCursor** function to save the current cursor position and text attributes. This function may only be called once for each time that the cursor position is saved.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetCursorPos, NvtSaveCursor, NvtSetCursorPos, NvtUpdateCaret, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSaveCursor Function

```
BOOL WINAPI NvtSaveCursor(
   HDISPLAY hDisplay
);
```

The **NvtSaveCursor** function saves the current cursor position and text attributes.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

Use the **NvtRestoreCursor** function to restore the cursor position and text attributes to their previous values.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetCursorPos, NvtRestoreCursor, NvtSetCursorPos, NvtUpdateCaret, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtScrollDisplay Function

```
BOOL WINAPI NvtScrollDisplay(
   HDISPLAY hDisplay,
   BOOL bScrollUp
);
```

The **NvtScrollDisplay** function scrolls the virtual display up or down and updates the scroll box position if necessary.

Parameters

hDisplay

Handle to the virtual display.

bScrollUp

Boolean flag which specifies if the virtual display is scrolled up or down.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. The **NvtRefreshDisplay** function should be called to update the window attached to the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayScrollPos, NvtRefreshDisplay, NvtSetDisplayScrollPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSelectDisplayText Function

```
BOOL WINAPI NvtSelectDisplayText(
    HDISPLAY hDisplay,
    LPRECT lpRect,
    DWORD dwOptions
);
```

The **NvtSelectDisplayText** function selects or unselects a region of the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpRect

A pointer to a region of the display to select. The coordinates must be in display cursor coordinates, not pixels. If this parameter is NULL, any selected text in the display is unselected.

dwOptions

One or more options which specifies how the region will be selected. These options may be combined using a bitwise Or operator. The following values may be used:

Value	Description
NVT_SELECT_DEFAULT	The default selection option. If there is a region of the display already selected, it will be cleared and the new region is selected.
NVT_SELECT_CLIPBOARD	Copy the selected text to the clipboard. If this option is not specified, the selected text is buffered and may be copied at a later point.
NVT_SELECT_NOREFRESH	The display is not refreshed when the region is selected. This is useful if the application is going to be selecting multiple regions of the display, or combining more than one region, in order to minimize output to the window.
NVT_SELECT_NOBUFFER	Do not buffer the text in the selected region of the display. The display will show any text as being selected, but it will not be available to be copied by the application. This can be useful if the application is going to select multiple regions and combine them.
NVT_SELECT_COMBINE	If there is already a region of the display that has been selected, the new region is combined with the previous region, selecting all of the text.
NVT_SELECT_UNSELECT	Unselect the specified region of the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCopySelectedText, NvtGetSelectedText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetCursorPos Function

```
BOOL WINAPI NvtSetCursorPos(
   HDISPLAY hDisplay,
   INT nCursorX,
   INT nCursorY
);
```

The **NvtSetCursorPos** function sets the current cursor column and row position on the virtual display.

Parameters

hDisplay

Handle to the virtual display.

nCursorX

New cursor column position. If this value is greater than the maximum number of columns, the current position is set to the last column on the display. The first column on the display is zero.

nCursorY

New cursor row position. If this value is greater than the maximum number of rows, the current position is set to the last row on the display. The first row on the display is zero.

Return Value

If the function succeeds, the return value is non-zero. If the handle to the virtual display is invalid, the function will return zero.

Remarks

The **NvtSetCursorPos** function sets the current cursor position on the virtual display. If the display is in origin mode (a scrolling region has been set), then the cursor row position is bound by the current region.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetCursorPos

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayAttributes Function

```
UINT WINAPI NvtSetDisplayAttributes(
    HDISPLAY hDisplay,
    UINT nAttributes
);
```

The **NvtSetDisplayAttributes** function sets the current display attributes.

Parameters

hDisplay

Handle to the virtual display.

nAttributes

An unsigned integer value which specifies the new display attributes. This may be one or more of the following values:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

Return Value

If the function succeeds, the return value is the previous display attributes. If the function fails, the return value is NVT_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayAttributes, NvtGetDisplayCell

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayBackColor Function

```
BOOL WINAPI NvtSetDisplayBackColor(
   HDISPLAY hDisplay,
   COLORREF rgbBackground
);
```

The **NvtSetDisplayBackColor** sets the background color used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

rgbBackground

The background color specified as a 32-bit RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function sets the background color for normal, reverse, bold and reverse-bold text attributes. To set the background color for a specific attribute, use the **NvtSetDisplayColor** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColor, NvtSetDisplayBoldColor, NvtSetDisplayColor, NvtSetDisplayForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayBoldColor Function

```
BOOL WINAPI NvtSetDisplayBoldColor(
    HDISPLAY hDisplay,
    COLORREF rgbBold
);
```

The **NvtSetDisplayBoldColor** sets the color used by the virtual display to display text with the bold attribute enabled.

Parameters

hDisplay

Handle to the virtual display.

rgbBold

The bold attribute color specified as a 32-bit RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function sets the foreground color for the bold and reverse-bold text attributes. To set the color for a specific attribute, use the **NvtSetDisplayColor** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColor, NvtSetDisplayBackColor, NvtSetDisplayColor, NvtSetDisplayForeColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayCell Function

```
DWORD WINAPI NvtSetDisplayCell(
    HDISPLAY hDisplay,
    INT xPos,
    INT yPos,
    DWORD dwCell
);
```

The **NvtSetDisplayCell** function sets the value of the specified cell in the virtual display.

Parameters

hDisplay

Handle to the virtual display.

xPos

An integer value which specifies the cell column in the virtual display. A value of -1 specifies that the current cursor position should be used.

yPos

An integer value which specifies the cell row in the virtual display. A value of -1 specifies that the current cursor position should be used.

dwCell

An unsigned integer which specifies the new value for the cell in the virtual display. The low order word of this value should contain the character to be displayed at that location. The high order word should specify the attributes for that cell.

Return Value

If the function succeeds, the return value is the previous cell value. If the function fails, it returns 0xFFFFFFFF. The previous cell value is the same value returned by the **NvtGetDisplayCell** function.

Remarks

The **NvtSetDisplayCell** function is used to modify a specific character cell in the virtual display, changing both the character and the attributes for that cell. Unlike the higher level functions such as **NvtWriteDisplay** which process character strings and escape sequences, the **NvtGetDisplayCell** and **NvtSetDisplayCell** functions allow direct, low-level access to the virtual display in memory. After one or more cells are modified using this function, you should call the **NvtRefreshDisplay** function to redraw the virtual display.

The character cell attributes may be one or more of the following values:

Value	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.

NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayAttributes, NvtGetDisplayCell, NvtRefreshDisplay, NvtSetDisplayAttributes, NvtWriteDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayColor Function

```
BOOL WINAPI NvtSetDisplayColor(
    HDISPLAY hDisplay,
    UINT nColorIndex,
    COLORREF dwColor,
    BOOL bForeground
);
```

The **NvtSetDisplayColor** function sets the colors used by the virtual display. Each display has a color table which specifies the foreground and background colors used when displaying text with a specific attribute.

Parameters

hDisplay

Handle to the virtual display.

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Value	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

dwColor

The RGB color value that will be used.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColor, NvtSetDisplayBackColor, NvtSetDisplayBoldColor, NvtSetDisplayForeColor

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayColorMap Function

```
BOOL WINAPI NvtSetDisplayColorMap(
    HDISPLAY hDisplay,
    COLORREF *lpColor,
    INT nColors
);
```

The **NvtSetDisplayColorMap** function modifies the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

hDisplay

Handle to the virtual display.

lpColor

A pointer to an array of COLORREF values which specifies the values to be used by the emulator when setting a color attribute. If this value is NULL, the default color table will be loaded.

nColors

The number of colors which are stored in the array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid or the number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **NvtSetDisplayColorMap** function is useful for changing what values are being used when a color attribute it set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
Black	RGB(0,0,0)			
Red	000000A0h	RGB(160,0,0)		
Green	0000A000h	RGB(0,160,0)		
Yellow	0000A0A0h	RGB(160,160,0)		
Blue	00A00000h	RGB(0,0,160)		
Magenta	00A000A0h	RGB(160,0,160)		
Cyan	00A0A000h	RGB(0,160,160)		
White	00E0E0E0h	RGB(224,224,224)		
Gray	00C0C0C0h	RGB(192,192,192)		
Light Red	008080FFh	RGB(255,128,128)		

Light Green	0090EE90h	RGB(144,238,144)
Light Yellow	00C0FFFFh	RGB(255,255,192)
Light Blue	00E6D8ADh	RGB(173,216,230)
Light Magenta	00FFC0FFh	RGB(255,192,255)
Light Cyan	00FFFFE0h	RGB(224,255,255)
High White	00FFFFFFh	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

The **NvtSetDisplayColorMap** function is used to modify the actual color displayed by the emulator. For example, if the emulator processes an escape sequence which sets the current foreground color to white, the actual color displayed could be changed to light green. Passing a NULL pointer as the second parameter restores the original color map back to the default values. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if ( NvtGetDisplayColorMap(hDisplay, rgbColor, 16) )
{
     rgbColor[7] = rgbColor[15];
     NvtSetDisplayColorMap(hDisplay, rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColor, NvtGetDisplayColorMap, NvtGetTextColor, NvtSetDisplayColor, NvtSetTextColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayDC Function

```
BOOL WINAPI NvtSetDisplayDC(
   HDISPLAY hDisplay,
   HDC hDC
);
```

The **NvtSetDisplayDC** function sets the device context to be used by the virtual display when updating the window.

Parameters

hDisplay

Handle to the virtual display.

hDC

Handle to the device context. This parameter may be NULL, any device context that is currently associated with the virtual display will be removed. Note that the application still has the responsibility for deleting the device context, otherwise a handle leak will occur.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

It is not required that the device context be explicitly set by the application. By default, the library will use a device context created using the window attached to the virtual display. If a device context is specified by the application, it must be released when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayDC, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayEmulation Function

```
BOOL WINAPI NvtSetDisplayEmulation(
    HDISPLAY hDisplay,
    UINT nEmulation
);
```

The **NvtSetDisplayEmulation** function specifies the type of terminal emulation to be performed by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

Changing the emulation type will not affect the current display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayEmulation, NvtGetDisplayMode, NvtResetDisplay, NvtSetDisplayMode

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayFocus Function

```
BOOL WINAPI NvtSetDisplayFocus(
    HDISPLAY hDisplay,
    BOOL bFocus
);
```

The **NvtSetDisplayFocus** function sets or removes the focus from the virtual display. This function should be called when the display window receives or loses focus.

Parameters

hDisplay

Handle to the virtual display.

bFocus

A boolean flag which specifies that the display should receive or lose focus.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

When the virtual display receives focus, it updates the current cursor position and displays the caret. When the display loses focus, the caret is hidden. This function should be called in response to the display window receiving the WM_SETFOCUS and WM_KILLFOCUS messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtUpdateCaret

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayFont Function

```
BOOL WINAPI NvtSetDisplayFont(
   HDISPLAY hDisplay,
   HFONT hFont
);
```

The **NvtSetDisplayFont** sets the font that will be used when updating the display. The specified font must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

Parameters

hDisplay

Handle to the virtual display.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

If the previous font was a default font created by the library as the result of a NULL font handle being passed to a function, it will be released. However, if the previous font was created by the application, the **DeleteObject** function must be called to release it.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayFont, NvtSetDisplayFontName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayFontName Function

```
BOOL WINAPI NvtSetDisplayFontName(
    HDISPLAY hDisplay,
    LPTSTR lpszFontName,
    INT nFontSize
);
```

The **NvtSetDisplayFontName** function sets the font that is to be used when updating the display. This function always attempts to load a font which is fixed-width and uses the OEM character set.

Parameters

hDisplay

Handle to the virtual display.

lpszFontName

A null-terminated string which specifies the name of the font that will be loaded. If a NULL pointer or an empty string is passed as the value, the default **Terminal** font will be used.

nFontSize

The point size of the font that will be loaded. If this value is zero, a default point size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayFont** function will return the handle to the font created by this function. However, this handle should not be released using the **DeleteObject** function. Font handles created by calling this function or passing a NULL handle to the **NvtSetDisplayFont** function will be automatically released by the library when the font is changed or the virtual display is destroyed.

If the previous display font was created using the **CreateFont** or **CreateFontIndirect** functions and then set using **NvtSetDisplayFont**, the font handle must be released by calling the **DeleteObject** function after this function has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtGetDisplayFont, NvtSetDisplayFont

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayForeColor Function

```
BOOL WINAPI NvtSetDisplayForeColor(
   HDISPLAY hDisplay,
   COLORREF rgbForeground
);
```

The **NvtSetDisplayForeColor** sets the foreground color used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

rgbForeground

The foreground color specified as a 32-bit RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function sets the foreground color for the normal and reverse text attributes. To set the foreground color for a specific attribute, use the **NvtSetDisplayColor** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColor, NvtSetDisplayBackColor, NvtSetDisplayBoldColor, NvtSetDisplayColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayMode Function

```
BOOL WINAPI NvtSetDisplayMode(
   HDISPLAY hDisplay,
   UINT nMode,
   BOOL bEnable
);
```

The **NvtSetDisplayMode** function sets one or more display modes for the specified virtual display.

Parameters

hDisplay

Handle to the virtual display.

nMode

The virtual display mode bitmask. This value is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other functions. The following values may be specified.

Value	Description
NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the foreground and background colors. This option is

	enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT (horizontal tab) control sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less then the total number of columns in the virtual display. Disabling this mode prevents a horizontal scroll bar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less then the total number of rows in the virtual display. Disabling this mode prevents a vertical scroll bar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtGetDisplayInfo, NvtGetDisplayMode, NvtSetDisplayEmulation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayScrollPos Function

```
BOOL WINAPI NvtSetDisplayScrollPos(
    HDISPLAY hDisplay,
    INT nScrollBar,
    INT nScrollPos
);
```

The **NvtSetDisplayScrollPos** function sets the position of the scroll box for the specified scroll bar and redraws the scroll bar to reflect the new position of the scroll box.

Parameters

hDisplay

Handle to the virtual display.

nScrollBar

Specifies the scroll bar to be set. This parameter can be one of the following values:

Value	Description
SB_HORZ	Sets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Sets the position of the scroll box in the display's standard vertical scroll bar.

nScrollPos

Specifies the new row or column of the scroll box. The position must be within the scrolling range.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function should always be used to change the scroll box position. The **NvtSetDisplayScrollPos** function will result in unpredictable behavior if used on the virtual display window.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayScrollPos, NvtScrollDisplay, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplaySize Function

```
BOOL WINAPI NvtSetDisplaySize(
   HDISPLAY hDisplay,
   LPSIZE lpSize
);
```

The **NvtSetDisplaySize** sets the size of the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpSize

Pointer to a SIZE structure which specifies the new size of the virtual display. The width should be specified in columns and the height should be specified in rows.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayInfo, NvtGetDisplayRect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetDisplayWindow Function

```
BOOL WINAPI NvtSetDisplayWindow(
    HDISPLAY hDisplay,
    HWND hWnd
);
```

The **NvtSetDisplayWindow** function sets the window used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

hWnd

Handle to the window that will be used by the virtual display. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtCreateDisplay, NvtGetDisplayDC, NvtGetDisplayWindow, NvtResetDisplay, NvtSetDisplayDC

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetMappedKey Function

```
BOOL WINAPI NvtSetMappedKey(
   HDISPLAY hDisplay,
   UINT nMappedKey,
   LPCTSTR lpszKeyBuffer
);
```

The **NvtSetMappedKey** function maps an escape sequence to the specified key.

Parameters

hDisplay

Handle to the virtual display.

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (function) keys for the current emulation.

lpszKeyBuffer

Pointer to a string which defines the sequence of characters that are to be mapped to the specified key. Passing an empty string or NULL pointer will delete the current sequence mapped to the key and restore the default value, if one has been defined.

Return Value

If the function succeeds, the return value is non-zero and the escape sequence is mapped to the specified key. If the key value is invalid or could not be mapped, then the function will return zero.

Remarks

The **NvtSetMappedKey** function maps an escape sequence to a special key. This function can be used to specify what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 function key). There are a number of default sequences that are mapped to the function and cursor keys, based on the current emulation. Calling this function will override the default sequence for a key, if one has been defined.

The following special keys are defined:

Value	Description
NVT_F1	F1 function key
NVT_F2	F2 function key
NVT_F3	F3 function key
NVT_F4	F4 function key
NVT_F5	F5 function key
NVT_F6	F6 function key
NVT_F7	F7 function key
NVT_F8	F8 function key
NVT_F9	F9 function key
NVT_F10	F10 function key

NVT_F11	F11 function key
NVT_F12	F12 function key
NVT_SF1	Shift F1 function key
NVT_SF2	Shift F2 function key
NVT_SF3	Shift F3 function key
NVT_SF4	Shift F4 function key
NVT_SF5	Shift F5 function key
NVT_SF6	Shift F6 function key
NVT_SF7	Shift F7 function key
NVT_SF8	Shift F8 function key
NVT_SF9	Shift F9 function key
NVT_SF10	Shift F10 function key
NVT_SF11	Shift F11 function key
NVT_SF12	Shift F12 function key
NVT_ENTER	Enter key
NVT_ERASE	Backspace key
NVT_UP	Cursor up key
NVT_DOWN	Cursor down key
NVT_LEFT	Cursor left key
NVT_RIGHT	Cursor right key
NVT_INSERT	Insert key
NVT_DELETE	Delete key
NVT_HOME	Home key
NVT_END	End key
NVT_PGUP	Page up key
NVT_PGDN	Page down key
NVT_APPUP	Up arrow key
NVT_APPDOWN	Down arrow key
NVT_APPLEFT	Left arrow key
NVT_APPRIGHT	Right arrow key
NVT_APPENTER	Keypad enter key
NVT_KEYPAD0	Numeric keypad 0
NVT_KEYPAD1	Numeric keypad 1
NVT_KEYPAD2	Numeric keypad 2
NVT_KEYPAD3	Numeric keypad 3

NVT_KEYPAD4	Numeric keypad 4
NVT_KEYPAD5	Numeric keypad 5
NVT_KEYPAD6	Numeric keypad 6
NVT_KEYPAD7	Numeric keypad 7
NVT_KEYPAD8	Numeric keypad 8
NVT_KEYPAD9	Numeric keypad 9

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtGetDisplayMode, NvtGetMappedKey, NvtResetMappedKeys, NvtSetDisplayMode, NvtTranslateMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetScrollRegion Function

```
BOOL WINAPI NvtSetScrollRegion(
    HDISPLAY hDisplay,
    INT nTop,
    INT nBottom
);
```

The **NvtSetScrollRegion** function sets the top and bottom rows of the current scrolling region.

Parameters

hDisplay

Handle to the virtual display.

nTop

The top row of the current scrolling region. If this value is greater than the bottom scrolling region row or the total number of rows in the display, it will be silently adjusted.

nBottom

The bottom row of the current scrolling region. If this value is less than zero or the top scrolling region row, it will be silently adjusted.

Return Value

If the function succeeds, it will return zero. If the handle to the display is invalid, the function will return zero.

Remarks

The **NvtSetScrollRegion** function allows an application to set the current scrolling region for the virtual display. This specifies the region (between the top and bottom rows) in which text will normally scroll. If the display is in origin mode, the cursor cannot be positioned outside of the scrolling region.

The minimum scrolling region that may be defined is two rows. If a scrolling region is specified that is less than two rows, the function will fail and the current scrolling region will remain unchanged. Specifying values of -1 for both arguments will reset the scrolling region to the default values (the full display).

The DEC STBM escape sequence is used to set or clear the scrolling region of the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayMode, NvtSetDisplayMode, NvtSetScrollRegion

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtSetTextColor Function

```
BOOL WINAPI NvtSetTextColor(
   HDISPLAY hDisplay,
   COLORREF rgbColor,
   BOOL bForeground
);
```

The **NvtSetTextColor** function changes the current foreground or background text color.

Parameters

hDisplay

Handle to the virtual display.

rgbColor

A color value which specifies the current foreground or background text color. The RGB macro can be used to specify the red, green and blue values for the color.

bForeground

A boolean value which determines if the foreground or background color is changed. A value of TRUE indicates that the foreground color should be changed, otherwise the background color is changed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. Failure indicates that the handle to the display is invalid or that the current display does not support color text attributes.

Remarks

This function is used to change the current foreground or background color, as determined by the text attribute. Note that changing the current foreground or background text color does not affect the virtual display color table. To change how color attributes are mapped to an RGB color value, use the **NvtSetDisplayColorMap** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayColorMap, NvtGetTextColor, NvtSetDisplayColorMap

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtTranslateMappedKey Function

```
BOOL WINAPI NvtTranslateMappedKey(
   HDISPLAY hDisplay,
   UINT nKey,
   UINT nFlags,
   UINT *lpnMappedKey,
   LPTSTR lpszKeyBuffer,
   UINT cchBuffer
);
```

The **NvtTranslateMappedKey** function translates a virtual key press to an escape sequence based on the current terminal emulation that has been selected.

Parameters

hDisplay

Handle to the virtual display.

nKey

The virtual key code for the specified key.

nFlags

The scan code, key-transition code, previous key state, and context code for the specified key.

lpnMappedKey

A pointer to an unsigned integer which will contain the index into the keymap table when the function returns. This is the same value used with the **NvtGetMappedKey** and **NvtSetMappedKey** function. If the index into the keymap is not required, this parameter can be NULL.

lpszKeyBuffer

Address of the buffer to receive the escape sequence mapped to the specified key. If the mapped key string is not required, this parameter can be NULL.

cchBuffer

The maximum number of characters that can be copied into the key buffer, including the terminating null character. If the ***lpszKeyBuffer*** parameter is NULL, this value must be zero.

Return Value

If the virtual key can be mapped to an escape sequence, the function will return a non-zero value. If the key is not mapped, or one of the arguments is invalid, the function will return zero.

Remarks

The **NvtTranslateMappedKey** function allows an application to map a virtual key code to an escape sequence that is appropriate for the type of terminal that is being emulated. For example, it will return the escape sequence for the F1 function key when passed the VK_F1 key value. This function should be called when the WM_KEYDOWN message is processed by an application so that it may send the correct sequence to the server.

This function should only be called in response to a keyboard message such as WM_KEYDOWN. To determine if a specific key has been mapped to an escape sequence, use the **NvtGetMappedKey** function.

Example

```
case WM_KEYDOWN:
    /*
     * If the Num Lock key is pressed, then set the terminal into application
     * keypad mode. This will change how the NvtTranslateMappedKey function will
     * translate the keypad keys.
     *
     * Note that the terminal may also be placed into application keypad mode
     * if emulating a DEC VT terminal and the DECNKM escape sequence is sent
     * by the host.
     */
    if (wParam == VK_NUMLOCK)
        NvtSetDisplayMode(hDisplay, NVT_MODE_APPKEYPAD, !(GetKeyState(VK_NUMLOCK) &
1));
    else
    {
        BOOL bMapped;
        UINT nMappedKey;
        TCHAR szKey[128];

        bMapped = NvtTranslateMappedKey(hDisplay, wParam, HIWORD(lParam),
                                        &nMappedKey, szKey, 128);

        if (bMapped)
            TelnetWrite(hClient, szKey, lstrlen(szKey));
    }
    break;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

NvtGetDisplayMode, NvtGetMappedKey, NvtResetMappedKeys, NvtSetDisplayMode, NvtSetMappedKey

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtUninitialize Function

```
VOID WINAPI NvtUninitialize();
```

The **NvtUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **NvtInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **NvtUninitialize** to allow the library to free any resources allocated on behalf of the process.

There should be a call to **NvtUninitialize** for every successful call to **NvtInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtDestroyDisplay, NvtUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtUpdateCaret Function

```
BOOL WINAPI NvtUpdateCaret(
   HDISPLAY hDisplay
);
```

The **NvtUpdateCaret** function updates the position of the caret in the display window to the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtSetCursorPos, NvtSetDisplayMode, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtUpdateDisplay Function

```
BOOL WINAPI NvtUpdateDisplay(
   HDISPLAY hDisplay
);
```

The **NvtUpdateDisplay** function updates the window attached to the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function should be called when the display window receives a WM_PAINT message. If the **NvtSetDisplayDC** function has not been called to explicitly set the display device context, this function will acquire one for the display window. In this case, the application should not create a device context for the window or call the **BeginPaint** and **EndPaint** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtRefreshDisplay, NvtResizeDisplay, NvtSetDisplayDC, NvtSetDisplayScrollPos, NvtSetDisplayWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NvtWriteDisplay Function

```
BOOL WINAPI NvtWriteDisplay(
   HDISPLAY hDisplay,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **NvtWriteDisplay** function writes the contents of the specified buffer to the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpBuffer

Pointer to the buffer which contains the data to be written to the virtual display.

cbBuffer

Number of bytes to write to the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function writes the data at the current cursor location. Control characters are recognized by this function and processed accordingly. If ANSI emulation is enabled, embedded escape sequences will also be parsed and processed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: csnvtv12.lib

See Also

NvtGetDisplayText, NvtRefreshDisplay, NvtUpdateCaret, NvtUpdateDisplay

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulation Data Structures

- NVTDISPLAYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

NVTDISPLAYINFO

The **NVTDISPLAYINFO** structure contains information about the virtual terminal display.

```
typedef struct _NVTDISPLAYINFO {
   HWND    hWnd;
   HFONT   hFont;
   INT     xPos;
   INT     yPos;
   INT     cxClient;
   INT     cyClient;
   INT     cxChar;
   INT     cyChar;
   INT     nScrollCol;
   INT     nScrollRow;
   INT     nMaxScrollCol;
   INT     nMaxScrollRow;
} NVTDISPLAYINFO, *LPNVTDISPLAYINFO;
```

Members

hWnd

The handle to the terminal emulation display window.

hFont

The handle to the current font.

xPos

The current display x coordinate.

yPos

The current display y coordinate.

cxClient

The width of the client window in pixels.

cyClient

The height of the client window in pixels.

cxChar

The width of the current font in pixels.

cyChar

The height of the current font in pixels.

nScrollCol

The current horizontal scrolling column.

nScrollRow

The current vertical scrolling row.

nMaxScrollCol

The maximum horizontal scrolling column.

nMaxScrollRow

The maximum vertical scrolling row.

Remarks

This structure is populated by the **NvtGetDisplayInfo** function to provide information about the current state of the virtual terminal. Member of this structure should be modified directly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

NvtCreateDisplay, NvtGetDisplayInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Terminal Emulator Control Sequences

Terminal Control Sequences

<ESC>c	Reset display to initial state
<ESC>8	Display alignment test

Cursor Control Sequences

<ESC>D	Move cursor down to next line
<ESC>E	Move cursor to first column and down one line
<ESC>M	Move cursor up one line
<ESC>7	Save cursor position, attributes and colors
<ESC>8	Restore saved cursor position, attributes and colors
<ESC>[*n*A	Move cursor up *n* lines
<ESC>[*n*B	Move cursor down *n* lines
<ESC>[*n*C	Move cursor forward *n* spaces
<ESC>[*n*D	Move cursor backward *n* spaces
<ESC>[*n*E	Move cursor to beginning of line, down *n* lines
<ESC>[*n*F	Move cursor to beginning of line, up *n* lines
<ESC>[*x*G	Move cursor to column *x*
<ESC>[*y*;*x*H	Move cursor to line *y*, column *x*
<ESC>[*n*I	Move cursor forward *n* tabstops
<ESC>[*n*Z	Move cursor backwards *n* tabstops
<ESC>[*n*a	Move cursor forward *n* spaces
<ESC>[*y*d	Move cursor to row *y*
<ESC>[*n*e	Move cursor down *n* lines
<ESC>[*y*;*x*f	Move cursor to line *y*, column *x*
<ESC>[s	Save cursor position
<ESC>[u	Return to saved cursor position
<ESC>[x`	Move cursor to column *x*

Attribute and Color Sequence

Select display attributes and color

n Value	Description
0	Reset to default attributes and colors
1	Bold attribute
2	Dim attribute
4	Underline attribute
5	Blink attribute (same as reverse)
7	Reverse attribute
8	Hidden attribute
22	Clear bold attribute
24	Clear underline attribute
25	Clear blink attribute

<ESC>[*n*m	27	Clear reverse attribute
	29	Clear color attributes
	30	Black foreground
	31	Red foreground
	32	Green foreground
	33	Yellow foreground
	34	Blue foreground
	35	Magenta foreground
	36	Cyan foreground
	37	White foreground
	40	Black background
	41	Red background
	42	Green background
	43	Yellow background
	44	Blue background
	45	Magenta background
	46	Cyan background
	47	White background

Character Set Sequences

<ESC>(A	Assign ISO Latin 1 character set to font bank G0
<ESC>(B	Assign United States ASCII character set to font bank G0
<ESC>(0	Assign graphics character set to font bank G0
<ESC>)A	Assign ISO Latin 1 character set to font bank G1
<ESC>)B	Assign United States ASCII character set to font bank G1
<ESC>)0	Assign graphics character set to font bank G1

Erase Sequences

<ESC>[*n*@ Insert ***n*** blank spaces

Erase all or part of the display

<ESC>[*n*J

n Value	Description
0	From current position to end of display
1	From beginning of display to current position
2	Erase the entire display

Erase all or part of a line

<ESC>[*n*K

n Value	Description
0	From current position to end of line
1	From beginning of line to current position
2	Erase the entire line

<ESC>[*n*L Insert ***n*** new blank lines

<ESC>[*n*M	Delete *n* lines from current cursor position
<ESC>[*n*P	Delete *n* characters from current cursor position

Scrolling Sequences

<ESC>[*n*S	Scroll display up *n* lines
<ESC>[*n*T	Scroll display down *n* lines
<ESC>[*n*X	Erase *n* characters from the current position
<ESC>[*y1*;*y2*r	Set scrolling region from lines *y1* to *y2*

Keypad Sequences

<ESC>=	Place keypad into applications mode
<ESC>>	Place keypad into numeric mode

Emulation Option Sequences

<ESC>[?*n*h

Set emulation option

n Value	Description
1	Enable cursor key application mode
2	Enable ANSI escape sequences
5	Reverse foreground and background colors
6	Enable origin mode
7	Enable auto-wrap mode
20	Enable linefeed/newline mode
25	Display caret
66	Place keypad in applications mode

<ESC>[?*n*1

Set emulation option

n Value	Description
1	Disable cursor key application mode
2	Enable VT52 escape sequences
5	Restore foreground and background colors
6	Disable origin mode
7	Disable auto-wrap mode
20	Disable linefeed/newline mode
25	Hide caret
66	Place keypad in numeric mode

Console Escape Sequences

<ESC>[=*n*A	Set the overscan color (ignored)
<ESC>[=*n1*;*n2*B	Set bell sound (parameters ignored)
<ESC>[=*n1*;*n2*C	Set the caret size

<ESC>[=*n*D

Set background color intensity

n Value	Description
0	Decrease background color intensity

	1	Increase background color intensity
<ESC>[=*n*E	Set blink vs. bold attribute (ignored)	
<ESC>[=*n*F	Set normal foreground color	
<ESC>[=*n*G	Set normal background color	
<ESC>[=*n*H	Set reverse foreground color	
<ESC>[=*n*I	Set reverse background color	
<ESC>[=*n*J	Set graphics foreground color	
<ESC>[=*n*K	Set graphics background color	

n Value	Description
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	High White

Control Character Sequences

<CTL>G	Ring audible bell, if enabled
<CTL>H	Move cursor one character backwards
<CTL>I	Move cursor forward to next tabstop
<CTL>J	Move cursor down to next line
<CTL>M	Move cursor to beginning of line
<CTL>N	Select G1 character set
<CTL>O	Select G0 character set
<CTL>Z	Abort current escape sequence
<DEL>	Erase and move cursor one character backwards

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Library

Send text messages to a mobile communications device using a gateway service.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSTXTV12.DLL
Version	12.0.3024.1286
LibID	3F750221-287C-4D42-AFD5-570CE6A41217
Import Library	CSTXTV12.LIB
Dependencies	None

Overview

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The Text Message API provides functions that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

This library has been designed to assist developers in sending text message notifications as part of their application. For example, it can be used to enable your software to automatically send notifications when a specific event occurs, such as an error condition. This library is not designed to be used with software that will send out a large number of text messages to many users, and there are limitations on the number of messages that may be sent to different phone numbers over a short period of time. Because many recipients must pay a fee for each text message they receive, text messages should only be sent to those who explicitly request them.

Applications should use an alternate method for sending text messages. Most mobile carriers have shut down their email-to-SMS gateways, making this method of sending text messages unreliable or unavailable. It is recommended that you transition to dedicated SMS service providers, which deliver messages over HTTPS-based APIs instead of email.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1

(SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Functions

Function	Description
SmsDisableTrace	Disable logging of network function calls
SmsEnableTrace	Enable logging of network function calls to a text file
SmsEnumProviders	Enumerate the available wireless service providers
SmsGetErrorString	Return a description for the specified error code
SmsGetFirstProvider	Return information about the first supported wireless service provider
SmsGetGateway	Return information about the gateway server for the specified phone number
SmsGetNextProvider	Return information about the next supported wireless service provider
SmsGetProvider	Return information about the wireless service provider for the specified phone number
SmsGetLastError	Return the last error code
SmsInitialize	Initialize the library and validates the specified license key at runtime
SmsSendMessage	Send a text message to the specified mobile device
SmsSetLastError	Set the last error code
SmsUninitialize	Terminate use of the library by the application

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsDisableTrace Function

```
BOOL WINAPI SmsDisableTrace();
```

The **SmsDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

SmsEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsEnableTrace Function

```
BOOL WINAPI SmsEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **SmsEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsEnumProviders Function

```
INT WINAPI SmsEnumProviders(
   LPSMSPROVIDER lpProviders,
   INT nMaxProviders,
   DWORD dwReserved
);
```

The **SmsEnumProviders** function enumerates the supported wireless service providers and populates an array of **SMSPROVIDER** structures.

Parameters

lpProviders

A pointer to an array of SMSPROVIDER structures that will be populated with information about each service provider. If this parameter is NULL, the function will return the number of service providers that are known.

nMaxProviders

An integer value which specifies the maximum number of service providers that may be enumerated by this function. If this parameter is zero, the function will return the number of service providers that are known. If the ***lpProviders*** parameter is NULL, this value should be zero.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the number of known service providers. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**. If the ***lpProviders*** parameter is not NULL and the ***nMaxProviders*** parameter indicates the array is not large enough to store all of the provider information, this function will fail with an error indicating that the buffer is too small.

Remarks

The **SmsEnumProviders** function is used to enumerate all of the supported service providers, populating an array of **SMSPROVIDER** structures that contains information about each provider, such as their name, domain, region of the country they service and the maximum message size they will accept. Typically this would be used to update a user interface control such as a listbox or drop-down combobox, enabling a user to select a preferred service provider.

Because some programming languages may not support arrays of structures in the same way that C/C++ does, the **SmsGetFirstProvider** and **SmsGetNextProvider** functions offer an alternative way to easily enumerate the available service providers.

To obtain information about a single service provider, use the **SmsGetProvider** function.

Example

```
SMSPROVIDER smsProviders[MAXPROVIDERS];
INT nProviders;

nProviders = SmsEnumProviders(smsProviders, MAXPROVIDERS, 0);
if (nProviders == SMS_ERROR)
{
    DWORD dwError = SmsGetLastError();
```

```
    _tprintf(_T("An error has occurred, error code %d\n"), (INT)LOWORD(dwError));
}
else
{
    for (INT nIndex = 0; nIndex < nProviders; nIndex++)
        _tprintf(_T("%s\n"), smsProviders[nIndex].szName);

    _tprintf(_T("%d providers returned\n"), nProviders);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsGetFirstProvider, SmsGetGateway, SmsGetNextProvider, SmsGetProvider, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsGetErrorString Function

```
INT WINAPI SmsGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **SmsGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **SmsGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsGetLastError, SmsSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsGetFirstProvider Function

```
BOOL WINAPI SmsGetFirstProvider(
   LPSMSPROVIDER lpProvider,
   LPDWORD lpdwToken,
);
```

The **SmsGetFirstProvider** function returns information about the first supported wireless service provider.

Parameters

lpProvider

A pointer to an SMSPROVIDER structure that will contain information about the service provider. This parameter cannot be NULL.

lpdwToken

A pointer to an unsigned integer value that is used when enumerating the service providers. The value will be initialized by this function and the modified by subsequent calls to the **SmsGetNextProvider** function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetFirstProvider** function is used in conjunction with **SmsGetNextProvider** to enumerate all of the supported wireless providers available to the client. These two functions can be used as an alternative to the **SmsEnumProviders** function, which may not be easily used with some programming languages because it populates an array of data structures rather than a single structure. The data that is returned is identical, the only difference is the method by which the service providers are enumerated.

The *lpdwToken* parameter is used to maintain context between multiple calls to the **SmsGetNextProvider** function. It should be treated as an opaque value and never modified directly by the application.

Example

```
SMSPROVIDER smsProvider;
DWORD dwToken;
BOOL bResult;

bResult = SmsGetFirstProvider(&smsProvider, &dwToken);
while (bResult)
{
    pListBox->AddString(smsProvider.szName);
    bResult = SmsGetNextProvider(&smsProvider, &dwToken);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsEnumProviders, SmsGetGateway, SmsGetNextProvider, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsGetGateway Function

```
INT WINAPI SmsGetGateway(
   LPCTSTR lpszPhoneNumber,
   LPCTSTR lpszProvider,
   LPSMSGATEWAY lpGateway
);
```

The **SmsGetGateway** function returns text message service information for a phone number.

Parameters

lpszPhoneNumber

A null-terminated string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpszProvider

A null-terminated string which specifies the preferred service provider for this telephone number. If the preferred service provider is unknown, this parameter can be NULL and the default provider will be selected.

lpGateway

A pointer to an SMSGATEWAY structure that will contain information about the text message gateway when the function returns. This includes information such as the name of the provider, the server that will accept text messages for this phone number, and the recipient address that should be used. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetGateway** function returns information about the service provider and mail gateway for a specific phone number, and can be used to determine if a given phone number is assigned to a mobile device capable of receiving text messages. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number along with information about its text message gateway service.

If the ***lpszProvider*** parameter is not NULL, this will identify a preferred provider for the phone number specified. In the United States and Canada, most wireless common carriers are required to provide wireless number portability (WNP) which allows a customer to continue to use their current phone number even if they switch to another service provider. This can result in a situation where a specific phone number is shown as allocated to one provider, but in actuality that user has switched to a different provider. For example, a user may have originally purchased a phone and service with AT&T and then later switched to Verizon, but decided to keep their phone number. In this case, if Verizon was not specified as the preferred provider, the library would attempt to send the message to the AT&T gateway, since that was the original provider who allocated the phone number.

For most applications, the correct way to handle the situation in which a user may have switched to a different service provider is to allow them to select an alternate service provider in your user interface.

For example, you could display a drop-down list of available service providers, populated using the **SmsEnumProviders** function. If they select a preferred provider, then you would pass that value to this function. If they do not, then specify a NULL pointer and the default provider will be selected.

This function sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this function is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this function multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsEnumProviders, SmsGetProvider, SMSGATEWAY

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsGetLastError Function

```
DWORD WINAPI SmsGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SmsSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **SmsGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **SmsSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CHANNEL or SMS_ERROR. Those functions which call **SmsSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

SmsGetErrorString, SmsSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsGetNextProvider Function

```
BOOL WINAPI SmsGetNextProvider(
   LPSMSPROVIDER lpProvider,
   LPDWORD lpdwToken,
);
```

The **SmsGetNextProvider** function returns information about the next supported wireless service provider.

Parameters

lpProvider

A pointer to an SMSPROVIDER structure that will contain information about the service provider. This parameter cannot be NULL.

lpdwToken

A pointer to an unsigned integer value that is used when enumerating the service providers. The value will be initialized by the **SmsGetFirstProvider** function and the modified by subsequent calls to this function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the last service provider has been enumerated or the function fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetNextProvider** function is used in conjunction with **SmsGetFirstProvider** to enumerate all of the supported wireless providers available to the client. These two functions can be used as an alternative to the **SmsEnumProviders** function, which may not be easily used with some programming languages because it populates an array of data structures rather than a single structure. The data that is returned is identical, the only difference is the method by which the service providers are enumerated.

The *lpdwToken* parameter is used to maintain context between multiple calls to this function. It should be treated as an opaque value and never modified directly by the application.

Example

```
SMSPROVIDER smsProvider;
DWORD dwToken;
BOOL bResult;

bResult = SmsGetFirstProvider(&smsProvider, &dwToken);
while (bResult)
{
    pListBox->AddString(smsProvider.szName);
    bResult = SmsGetNextProvider(&smsProvider, &dwToken);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

SmsEnumProviders, SmsGetGateway, SmsGetNextProvider, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsGetProvider Function

```
INT WINAPI SmsGetProvider(
   LPCTSTR lpszPhoneNumber,
   LPSMSPROVIDER lpProvider
);
```

The **SmsGetProvider** function returns information about the service provider for the specified phone number.

Parameters

lpszPhoneNumber

A null-terminated string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpProvider

A pointer to an SMSPROVIDER structure that will contain information about the service provider for the specified phone number.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetProvider** function returns information about the service provider associated with a phone number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to the **SmsGetGateway** function.

This function sends an HTTP query to the server **api.sockettools.com** to obtain information about the wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this function is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this function multiple times using the same phone number will not increase the request count.

For a list of all supported wireless service providers, use the **SmsEnumProviders** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsEnumProviders, SmsGetGateway, SMSPROVIDER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsInitialize Function

```
BOOL WINAPI SmsInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **SmsInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmsGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **SmsInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **SmsUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **SmsInitialize** or **SmsUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsSendMessage, SmsUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsSendMessage Function

```
INT WINAPI SmsSendMessage(
   LPSMSSERVICE lpService,
   LPSMSMESSAGE lpMessage,
   DWORD dwReserved,
);
```

The **SmsSendMessage** function sends a text message to the specified mobile device.

Parameters

lpService

A pointer to an SMSSERVICE structure that identifies the messaging service that will be used to send the text message. The default service sends the message through the mail server gateway for the wireless service provider associated with the recipient's phone number. This parameter cannot be NULL.

lpMessage

A pointer to an SMSMESSAGE structure that contains information about the message to be sent, including the sender, the recipient and the text message itself. This parameter cannot be NULL.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsSendMessage** function is used to send a text message to a mobile device. This API is designed to support multiple methods of sending text messages, with the default method sending the message through a server gateway established by the wireless service provider.

SMS_SERVICE_SMTP

This message service sends the message through the wireless service provider's mail gateway using the SMTP protocol. However, it is important to note that many of these gateways will not accept messages from a client that is connected to them using a residential Internet service provider. If the application is being run on a system that uses a residential provider, that service provider may also block outbound connections to all mail servers other than their own. These anti-spam measures typically require that most end-user applications specify a relay mail server rather than submitting the message directly to the wireless provider's gateway.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to the **SmsGetGateway** function.

This service also sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per

minute. If this function is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this function multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsGetGateway, SmsGetProvider, SMSMESSAGE, SMSSERVICE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsSetLastError Function

```
VOID WINAPI SmsSetLastError(
   DWORD dwErrorCode
);
```

The **SmsSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMS_ERROR. Those functions which call **SmsSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **SmsGetLastError** function. The use of **SmsGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

SmsGetErrorString, SmsGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SmsUninitialize Function

```
VOID WINAPI SmsUninitialize();
```

The **SmsUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **SmsInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **SmsUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **SmsUninitialize** for every successful call to **SmsInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cstxtv12.lib

See Also

SmsInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Text Message Data Structures

- SMSGATEWAY
- SMSMESSAGE
- SMSPROVIDER
- SMSSERVICE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSGATEWAY

This structure contains information about a text message gateway server and the recipient.

```
typedef struct _SMSGATEWAY
{
    INT     nGatewayId;
    INT     nCountryCode;
    INT     nAreaCode;
    INT     nExchange;
    INT     nMessageLength;
    DWORD   dwReserved;
    TCHAR   szProvider[SMS_MAXPROVIDERNAMELEN];
    TCHAR   szDomain[SMS_MAXDOMAINNAMELEN];
    TCHAR   szServer[SMS_MAXMAILSERVERLEN];
    TCHAR   szAddress[SMS_MAXMAILADDRESSLEN];
}  SMSCHANNEL, *LPSMSCHANNEL;
```

Members

nGatewayId

An integer value which identifies the gateway record. This value is used internally and an application should not depend on the value not changing over time for a specific telephone number. As new area codes are introduced, the provider database will be updated to reflect these changes and that can result in a change to the gateway ID associated with a specific telephone number.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nAreaCode

An integer value which specifies the Numbering Plan Area (NPA) code, commonly known as the area code. For the United States and Canada, area codes are assigned by the North American Numbering Plan Administration (NANPA). In North America, the area code is digits 1-3 for a 10-digit telephone number. This value, along with the exchange, is used to determine which company provides wireless service for a specific telephone number.

nExchange

An integer value which specifies the exchange area. In North America, the exchange is digits 4-6 for a 10-digit telephone number. This value, along with the area code, is used to determine which company provides wireless service for a specific telephone number.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwReserved

A value reserved for internal use.

szProvider

A null-terminated string which identifies the name of the service provider that is associated with the specified telephone number. Note that this value may not represent the actual company that is providing the wireless service.

szDomain

A null-terminated string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

szServer

A null-terminated string which identifies the host name or IP address of the mail server used to accept text messages for the specified service provider. In some cases, a provider may have multiple gateway servers and this value will represent the preferred mail server for the domain.

szAddress

A null-terminated string which contains the complete email address that should be used when sending the text message through the gateway mail server. Different service providers can have slightly different rules about how the address is formatted, but it typically is a combination of the telephone number and the gateway domain name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSMESSAGE

This structure provides information about a text message.

```
typedef struct _SMSMESSAGE
{
    DWORD dwFormat;
    DWORD dwLength;
    DWORD dwFlags;
    DWORD dwReserved;
    LPCTSTR lpszProvider;
    LPCTSTR lpszPhoneNumber;
    LPCTSTR lpszSender;
    LPCTSTR lpszMessage;
}  SMSMESSAGE, *LPSMSMESSAGE;
```

Members

dwFormat

An integer value which specifies the format of the message message. This member is included for future use where a service provider supports multiple message formats based on different versions of the protocol. The default value for this member is SMS_FORMAT_TEXT.

dwLength

An integer value which specifies the length of the message. If this member is zero, the length will be automatically calculated based on the length of the ***lpszMessage*** text that is terminated by a null character. If this value is larger than the actual length of the message text, it will be ignored.

dwFlags

An integer value which specifies one or more message options.

Value	Description
SMS_MESSAGE_DEFAULT	The default value used with standard text messages.
SMS_MESSAGE_URGENT	The text message should be flagged as urgent. For messages that are sent through a mail gateway, this will set the header to indicate that it is a high priority message. Note that service providers handle urgent messages differently and some may ignore the message priority.

dwReserved

Reserved for future use. This value should always be zero.

lpszProvider

A null-terminated string which specifies the name of the preferred wireless service provider responsible for handling the message. If this member is NULL or an empty string, the default provider assigned to the recipient's phone number will be used. This structure member is only used with SMS_SERVICE_SMTP messages and is ignored for other message services.

lpszPhoneNumber

A null-terminated string which specifies the recipient's phone number. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. This structure member cannot be NULL.

lpszSender

A null-terminated string which identifies the sender of the message. For SMS_SERVICE_SMTP messages, this string should be a valid email address. For other services, this string may specify a phone number or shortcode. This structure member cannot be NULL.

lpszMessage

A null-terminated string that contains the message to be sent to the recipient. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsSendMessage, SMSSERVICE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSPROVIDER

This structure contains information about a wireless service provider.

```
typedef struct _SMSPROVIDER
{
    INT     nProviderId;
    INT     nCountryCode;
    INT     nRegionCode;
    INT     nMessageLength;
    DWORD   dwFlags;
    DWORD   dwReserved;
    TCHAR   szGuid[SMS_MAXPROVIDERGUIDLEN];
    TCHAR   szName[SMS_MAXPROVIDERNAMELEN];
    TCHAR   szCompany[SMS_MAXCOMPANYNAMELEN];
    TCHAR   szDomain[SMS_MAXDOMAINNAMELEN];
}  SMSPROVIDER, *LPSMSPROVIDER;
```

Members

nProviderId

An integer value which identifies the provider record. This value is used internally and an application should not depend on the value not changing over time for a specific service provider. To uniquely identify a provider, use the ***szGuid*** member of the structure, which is guaranteed not to change as providers are added and removed from the database.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nRegionCode

An integer value which identifies the region that the service provider covers. In North America, each region consists of multiple states and/or provinces. If a provider services multiple regions, this will identify the primary region where they provide coverage.

Value	Description
SMS_REGION_NATIONAL 0	All regions. This region code is used for service providers that have national coverage and do not exclusively provide service within one or more specific geographical regions.
SMS_REGION_NORTH_EAST_ATLANTIC 1	Northeastern Atlantic region which includes Connecticut, Maine, Massachusetts, New Hampshire, New Brunswick, Newfoundland, Nova Scotia, Rhode Island and Vermont.
SMS_REGION_MIDDLE_ATLANTIC 2	Middle Atlantic region which includes New Jersey, New York, Delaware, District of Columbia, Maryland, Pennsylvania, Quebec, Virginia and West Virginia.
SMS_REGION_EAST_NORTH_CENTRAL 3	Northeastern central region which includes Illinois, Indiana, Michigan, Ohio and Wisconsin.

SMS_REGION_SOUTH_ATLANTIC 4	Southern Atlantic region which includes Florida, Georgia, North Carolina and South Carolina.
SMS_REGION_EAST_SOUTH_CENTRAL 5	Southeastern central region which includes Alabama, Kentucky, Mississippi and Tennessee.
SMS_REGION_WEST_NORTH_CENTRAL 6	Northwestern central region which includes Iowa, Kansas, Manitoba, Minnesota, Missouri, Nebraska, North Dakota, Ontario and South Dakota.
SMS_REGION_WEST_SOUTH_CENTRAL 7	Southwestern central region which includes Arkansas, Louisiana, Oklahoma and Texas.
SMS_REGION_MOUNTAIN 8	Mountain region which includes Alberta, Arizona, Colorado, Idaho, Montana, Nevada, New Mexico, Northwest Territories, Saskatchewan, Utah and Wyoming.
SMS_REGION_PACIFIC 9	Pacific region which includes Alaska, British Columbia, California, Hawaii, Oregon, Washington and Yukon.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwFlags

An unsigned integer value which specifies one or more flags that provides additional information about the service provider. This value is constructed by using a bitwise operator with any of the following constants:

Value	Description
SMS_PROVIDER_DEFAULT 0	A standard service provider. Typically this means that customers have a service contract for their mobile device and pay monthly access and service charges.
SMS_PROVIDER_PREPAID 1	A service provider that offers pre-paid calling cards or fixed month-to-month payments that do not require long-term service contracts.

dwReserved

A value reserved for internal use.

szGuid

A null-terminated string which uniquely identifies the service provider. The string is in a standard format used for globally unique identifiers (GUIDs) and is guaranteed to not change for the service provider it has been assigned to.

szName

A null-terminated string which specifies the name of service provider. Note that this value may not represent the actual company that is providing the wireless service.

szCompany

A null-terminated string which specifies the name of the company associated with the service provider. This may be the same as name of the service provider itself or it may be the name of a parent company that owns the service provider.

szDomain

A null-terminated string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SMSSERVICE

This structure provides information about the service used to send a text message.

```
typedef struct _SMSSERVICE
{
    DWORD dwServiceType;
    DWORD dwAuthType;
    DWORD dwVersion;
    DWORD dwTimeout;
    DWORD dwOptions;
    DWORD dwReserved;
    LPCTSTR lpszResource;
    LPCTSTR lpszAccount;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
}  SMSSERVICE, *LPSMSSERVICE;
```

Members

dwServiceType

An integer value which identifies the type of service that will be used to send the message. This member can be one of the following values:

Value	Description
SMS_SERVICE_SMTP	The text message will be sent through the mail gateway for the specified service provider. This service uses SMTP to submit the message for delivery, either directly to the server provider's mail gateway server or through a relay server. This is the default service type.

dwAuthType

An integer value which identifies the type of authentication used with the service. This member can be one of the following values:

Value	Description
SMS_AUTH_DEFAULT	The default authentication method for this service type should be used. Most applications should use this value unless a service type provides multiple authentication methods.
SMS_AUTH_USERNAME	The service requires authentication using a username and password. This value can be used with an SMTP service that requires user authentication and is typically needed when using a mail server relay. For the SMS_SERVICE_SMTP service, this is the default authentication method.

dwVersion

An integer value which identifies the interface version for the service being used. This member is included for future use where a service may support multiple versions of their interface and should normally be set to the value SMS_VERSION_DEFAULT.

dwTimeout

An integer value which specifies the amount of time in seconds that a function will wait for a response from the service. If this value is zero, a default timeout period of 20 seconds will be used.

If the service does not respond within this time period, the function will fail.

dwOptions

An integer value which specifies one or more options.

Value	Description
SMS_OPTION_NONE	No additional options for the service.
SMS_OPTION_SECURE	This option specifies that TLS will be used to establish a secure, encrypted connection with the service. For some services, it may be required to connect to them securely and this option will be enabled automatically.

dwReserved

Reserved for future use. This value should always be zero.

lpszResource

A null-terminated string that specifies a resource for the service. Typically this will be either a fully qualified domain name or a URL. For gateways using SMTP, this string should identify the mail server. An alternate port number can also be specified by appending it to the hostname, separated by a colon. For example, **smtp.company.com:587** would connect to the server on port 587. If you are specifying an IPv6 address with an alternate port number, the address must be enclosed in brackets. For services where a domain name or resource URL is not required, this member will be ignored and can be NULL.

lpszAccount

A null-terminated string that specifies an account name or identifier. Some service providers may require a unique account name or a token (application ID) in conjunction with other credentials. This member is not used with mail gateways and ignored if the service type is SMS_SERVICE_SMTP. If no account name is required for session authentication, this member can be NULL.

lpszUserName

A null-terminated string that specifies a user name to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid user name. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "username" with their documentation; this member will always specify the first of a pair of authentication tokens.

lpszPassword

A null-terminated string that specifies the password used to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid password. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "password" with their documentation; this member will always specify the second of a pair of authentication tokens.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

See Also

SmsSendMessage, SMSMESSAGE

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Web Services Library

The Web Services library provides data storage and location services for applications.

Reference

- Functions
- Constants
- Data Structures
- Error Codes

Library Information

File Name	CSWEBV12.DLL
Version	12.0.3024.1286
LibID	ECC96063-0D95-4378-A747-E0040B8618F1
Import Library	CSWEBV12.LIB
Dependencies	None

Overview

The Web Services library enables an application to store and manage data remotely, and return information about the current physical location of the client. These functions use secure services provided by SocketTools API servers and do not require third-party party APIs or accounts with other cloud service providers.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most

commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Web Services Functions

Function	Description
WebCancelTransfer	Cancel a storage operation that is currently in progress
WebCloseStorage	Close a storage container and release resources allocated for the client session
WebCompareFile	Compare the contents of a stored object with a local file
WebCompareObject	Compare the contents of a stored object with a memory buffer
WebCompareText	Compare the contents of a stored object with a string
WebCopyObject	Copy a storage object to a new location and optionally rename the object label
WebDeleteObject	Delete an existing storage object
WebDownloadFile	Download the contents of a stored object to a local file
WebDownloadFileEx	Download the contents of a stored object with progress notifications
WebDisableTrace	Disable logging of function calls
WebEnableTrace	Enable logging of function calls to a file
WebEnumApplications	Enumerate all registered storage applications
WebEnumObjects	Enumerate all storage objects that match the specified label or content type
WebGetAccountId	Return the current web services account identifier
WebGetErrorString	Return a description for the specified error code
WebGetFile	Download the contents of a storage object and copy to a local file
WebGetFileEx	Download the contents of a storage object and return information about the object
WebGetFirstApplication	Return information about the first registered storage application
WebGetFirstObject	Return information about the first object that matches search criteria
WebGetLastError	Return the last error code
WebGetLocation	Return the current physical location of the local computer system
WebGetNextApplication	Return information about the next registered storage application
WebGetNextObject	Return information about the next object that matches search criteria
WebGetObject	Download the contents of a storage object to a buffer
WebGetObjectEx	Download the contents of a storage object to buffer and return information about the object
WebGetObjectInformation	Retrieve the metadata for the specified storage object
WebGetObjectSize	Return the size of the specified storage object
WebGetObjectTime	Return the creation and last modification time for the specified storage object
WebGetPosixTime32	Return a POSIX timestamp value as a 32-bit integer
WebGetPosixTime64	Return a POSIX timestamp value as a 64-bit integer
WebGetStorageId	Return the returns the storage container ID
WebGetStorageQuota	Return quota limits assigned to your storage account
WebGetStorageTimeout	Get the number of seconds until a storage operation times out
WebGetSystemTime	Return the current date and time in a SYSTEMTIME structure
WebGetTextObject	Download the contents of a text object into a string buffer

WebGetTransferStatus	Return status information about the progress of a data transfer
WebInitialize	Initialize the library and validates the specified license key at runtime
WebMoveObject	Move a storage object to a new location and optionally rename the object label
WebOpenStorage	Open a storage container and return a handle for the client session
WebPutFile	Upload the contents of a local file to a new storage object
WebPutFileEx	Upload the contents of a local file and return information about the new object
WebPutObject	Upload the contents of a memory buffer to a new storage object
WebPutObjectEx	Upload the contents of a memory buffer and return information about the new object
WebPutTextObject	Create or replace a text object with the contents of a string buffer
WebRegisterAppId	Register a new application identifier used to store and retrieve data
WebRegisterEvent	Register an event handler to receive notifications for the session
WebRenameObject	Change the label associated with a storage object
WebResetStorage	Resets the application storage container and deletes all stored objects
WebResetStorageEx	Resets the specified storage container for an application and deletes all stored objects
WebSetLastError	Set the last error code
WebSetStorageTimeout	Set the number of seconds until a storage operation times out
WebUninitialize	Terminates the use of the library
WebUnregisterAppId	Unregister the application identifier and delete all associated storage objects
WebUploadFile	Upload the contents of a local file, creating or overwriting a stored object
WebUploadFileEx	Upload the contents of a local file with progress notifications
WebValidateAppId	Validate the specified application identifier
WebValidateLabel	Check the specified string to ensure it is a valid object label

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebCancelTransfer Function

```
BOOL WINAPI WebCancelTransfer(
   HSTORAGE hStorage
);
```

The **WebCancelTransfer** function cancels the current data transfer in progress.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCancelTransfer** function will cancel the current data transfer and abort the connection to the storage server. This function will fail if an active data transfer (either an upload or download) is not in progress.

This would typically be used within an event handler to cancel an operation as the contents of an object are being read or written. There is no mechanism to resume a canceled data transfer and this function should only be used when absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetTransferStatus, WebRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebCloseStorage Function

```
BOOL WINAPI WebCloseStorage(
   HSTORAGE hStorage
);
```

The **WebCloseStorage** function closes the storage container.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCloseStorage** function must be called after all operations using the storage container have completed. The access token granted to the application will be released and the memory allocated for the session cache will be freed. Failure to call this function can result in a memory leak.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)
Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebOpenStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebCompareFile Function

```
BOOL WINAPI WebCompareFile(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszLocalFile
);
```

The **WebCompareFile** function compares the contents of a stored object with a local file.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string which specifies the label of the object to be compared.

lpszLocalFile

A null-terminated string that specifies the name of the local file. If a path is not specified, the file will be created in the current working directory.

Return Value

If the function succeeds, the return value is a non-zero and the contents of the file matches the stored object. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCompareFile** function performs a binary comparison of the contents of a local file with a stored object on the server. The contents of the file must be identical to the contents of the stored object or the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCompareObject, WebCompareText, WebGetObjectInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebCompareObject Function

```
BOOL WINAPI WebCompareObject(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPCVOID lpvBuffer,
   DWORD dwLength
);
```

The **WebCompareObject** function compares the contents of a stored object with the data provided by the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string which specifies the label of the object to be compared.

lpvBuffer

A buffer that contains the data to be compared against the storage object.

dwLength

An unsigned integer which specifies the length of the data buffer to be compared.

Return Value

If the function succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCompareObject** function performs a binary comparison of the data in the specified buffer with the contents of the storage object on the server. The ***dwLength*** parameter must match the size of the stored object exactly, or this function will fail. Partial comparisons are not supported by this function.

If you wish to compare the contents of a text object, it is recommended that you use **WebCompareText**. This function ensures that Unicode text is compared correctly.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCompareFile, WebCompareText, WebGetObjectInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebCompareText Function

```
BOOL WINAPI WebCompareText(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszObjectText,
   INT cchObjectText
);
```

The **WebCompareText** function compares the contents of a stored object with the string provided by the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string which specifies the label of the object to be compared.

lpszObjectText

A null-terminated string which contains the text to be compared with the stored object.

cchObjectText

The number of characters in the ***lpszObjectText*** string to be compared. This value may be -1, in which case the string length will be determined by counting the number of characters up to the terminating null.

Return Value

If the function succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCompareText** function performs a text comparison of the characters in the string with the contents of the storage object on the server. The string length must match the amount of text in the stored object exactly, or this function will fail. Partial comparisons are not supported by this function.

The Unicode version of this function will automatically convert the UTF-16 string to UTF-8 encoding in the same way the **WebPutTextObject** function does.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCompareFile, WebCompareObject, WebGetObjectInformation, WebGetTextObject, WebPutTextObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebCopyObject Function

```
BOOL WINAPI WebCopyObject(
  HSTORAGE hStorage,
  LPCTSTR lpszOldLabel,
  LPCTSTR lpszNewLabel,
  DWORD dwStorageType,
  LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebCopyObject** function creates a copy of an existing storage object using a new label.

Parameters

hStorage

A handle to the storage container.

lpszOldLabel

A null-terminated string which specifies the label of the existing storage object to be copied. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A null-terminated string which specifies the name of the new storage object that will be created. This parameter may be NULL or point to an empty string, in which case the label name is not changed. In this case, the ***dwStorageType*** parameter cannot be WEB_STORAGE_DEFAULT.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Value	Description
WEB_STORAGE_DEFAULT	The default storage type. If this value is specified, the new object will be created using the same storage type as the original storage object.
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously

	stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the new storage object. If this information is not required, this parameter may be a NULL pointer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCopyObject** function is used to create a copy of an existing storage object. It may be used to duplicate an object with a different label, or it may be used to copy the object to a new storage container type. For example, it can copy an object originally created using WEB_STORAGE_USER to a new object stored using WEB_STORAGE_MACHINE.

Copied objects are assigned their own unique ID and are not linked to one another. Any subsequent changes made to the original object will not affect the copied object. Attempting to copy an object to itself or another existing object will result in an error.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDeleteObject, WebMoveObject, WebRenameObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebDeleteObject Function

```
BOOL WINAPI WebDeleteObject(
  HSTORAGE hStorage,
  LPCTSTR lpszObjectLabel
);
```

The **WebDeleteObject** function deletes an object from the storage container.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string which specifies the label of the object to be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebDeleteObject** function permanently deletes the storage object and its associated data from the server. Deleted objects cannot be recovered by the application. To remove all objects stored in the container, use the **WebResetStorage** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCopyObject, WebMoveObject, WebRenameObject, WebResetStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebDisableTrace Function

```
BOOL WINAPI WebDisableTrace();
```

The **WebDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, a value of zero is returned.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebDownloadFile Function

```
BOOL WINAPI WebDownloadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebDownloadFile** function downloads the contents of a storage object and copies it to a local file.

Parameters

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server. This parameter cannot be NULL or an empty string. Refer to the **WebValidateLabel** function for more information about object labels.

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebDownloadFile** function downloads the contents of the storage object and stores it in a local file. The object must have been previously created in the storage container WEB_STORAGE_GLOBAL using the application ID `SocketTools.Storage.Default`. This is the same default container used by the **WebUploadFile** function.

Unlike the **WebGetFile** and **WebGetFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebDownloadFile**.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

The **WebDownloadFileEx** function provides similar functionality with a more complex interface that supports custom application IDs, additional options and the ability to provide a callback function that is invoked during the download process.

Example

```
WEB_STORAGE_OBJECT webObject;

// Download the object from global storage to a local file
if (WebDownloadFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
```

```
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be downloaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to download \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFileEx, WebGetFile, WebUploadFile, WebValidateLabel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebDownloadFileEx Function

```
BOOL WINAPI WebDownloadFileEx(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszAppId,
   DWORD dwStorageType,
   DWORD dwReserved,
   DWORD dwTimeout,
   LPWEB_STORAGE_OBJECT lpObject,
   WEBEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **WebDownloadFileEx** function downloads the contents of a storage object and copies it to a local file.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A null-terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.

WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

dwTimeout

An unsigned integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WebEventProc** callback function. If this parameter is NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback function specified by ***lpEventProc***. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebDownloadFileEx** function downloads the contents of the storage object and stores it in a local file. The **WebDownloadFile** function provides a simpler interface that defaults to downloading an object from the global storage container.

Unlike the **WebGetFile** and **WebGetFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebDownloadFileEx**.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are downloading a large object and want your application to receive progress updates during the data transfer, provide a pointer to a callback function as the ***lpEventProc*** parameter. That function will receive event notifications as the data is being downloaded.

Example

```
WEB_STORAGE_OBJECT webObject;

// Download the object from global storage to a local file
if (WebDownloadFileEx(lpszLocalFile,
                      lpszObjectLabel,
                      lpszAppId,
                      WEB_STORAGE_GLOBAL,
                      0,
                      &webObject,
                      NULL, 0))
{
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be retrieved, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to retrieve \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFileEx, WebPutFileEx, WebUploadFile, WebUploadFileEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebEnableTrace Function

```
BOOL WINAPI WebEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **WebEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebEnumApplications Function

```
BOOL WINAPI WebEnumApplications(
   LPWEB_STORAGE_APPLICATION lpApplications,
   LPDWORD lpdwRegistered
);
```

The **WebEnumApplications** function enumerates all registered applications associated with the storage account.

Parameters

lpApplications

A pointer to an array of WEB_STORAGE_APPLICATION structures that will contain information about the enumerated application IDs. If this parameter is NULL, then no information is returned.

lpdwRegistered

A pointer to an unsigned integer that will contain the number of registered applications enumerated by this function. If the ***lpApplications*** parameter points to an array of WEB_STORAGE_APPLICATION structures, this parameter must be initialized to the maximum size of the array being passed to the function. If the ***lpApplications*** parameter is NULL, this value must be initialized to zero and when the function returns it will contain the number of registered applications.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebEnumApplications** function can be used to enumerate the application IDs registered with the storage account using the **WebRegisterAppId** function.

This function can be used in two ways. If the ***lpApplications*** parameter is NULL and the value passed by reference in the ***lpdwRegistered*** parameter is initialized to zero, the function will return the number of registered applications in ***lpdwRegistered***. This can be used to dynamically determine the size of the ***lpApplications*** array instead of declaring a fixed-size array in your application.

If the ***lpApplications*** parameter is not NULL, then the value referenced by ***lpdwRegistered*** must be initialized to the maximum size of the array that ***lpApplications*** points to. It is important to note that if either parameter is not initialized correctly, it can result in memory corruption and/or an unhandled exception.

The **WebGetFirstApplication** and **WebGetNextApplication** functions can be used to iterate through all registered application IDs without requiring you to preallocate memory for an array. This can be more efficient if a large number of application IDs have been registered.

Example

```
DWORD dwAppCount = 0;

if (WebEnumApplications(NULL, &dwAppCount ))
{
    LPWEB_STORAGE_APPLICATION lpAppArray = NULL;
    DWORD dwIndex;
```

```
    // Return if there are no registered applications
    if (dwAppCount == 0)
        return;

    // Allocate memory for an array of WEB_STORAGE_APPLICATION structures
    lpAppArray = (LPWEB_STORAGE_APPLICATION)LocalAlloc(LPTR,
sizeof(WEB_STORAGE_APPLICATION) * dwAppCount);

    if (lpAppArray == NULL)
        return; // Exhausted virtual memory?

    // Enumerate the registered application IDs
    if (!WebEnumApplications(lpAppArray, &dwAppCount))
        return;

    // Print information about each application
    for (dwIndex = 0; dwIndex < dwAppCount ; dwIndex++)
    {
        _tprintf(_T("AppId: %s\n"), lpAppArray[dwIndex].szAppId);
        _tprintf(_T("Key: %s\n"), lpAppArray[dwIndex].szApiKey);
        _tprintf(_T("LUID: %s\n"), lpAppArray[dwIndex].szLuid);
        _tprintf(_T("Tokens: %lu\n"), lpAppArray[dwIndex].dwTokens);

        if (dwAppCount > 1 && dwIndex < dwAppCount - 1)
            _tprintf(_T("\n"));
    }

    LocalFree((HLOCAL)lpAppArray);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetFirstApplication, WebGetNextApplication, WebRegisterAppId, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebEnumObjects Function

```
BOOL WINAPI WebEnumObjects(
   HSTORAGE hStorage,
   LPCTSTR lpszMatchLabel,
   LPCTSTR lpszContentType,
   DWORD dwReserved,
   LPWEB_STORAGE_OBJECT lpObjects,
   LPDWORD lpdwObjects
);
```

The **WebEnumObjects** function enumerates all storage objects that match the specified label or content type.

Parameters

hStorage

A handle to the storage container.

lpszMatchLabel

A null-terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this value is a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A null-terminated string which specifies the content type of the objects to be enumerated. If this value is a NULL pointer or an empty string, the content type is ignored and all matching objects are returned.

dwReserved

An unsigned integer value that is reserved for future use. This value must be zero.

lpObjects

A pointer to an array of WEB_STORAGE_OBJECT structures that will contain information about the enumerated objects. If this parameter is NULL, then no object information is returned.

lpdwObjects

A pointer to an unsigned integer that will contain the number of objects enumerated by this function. If the ***lpObjects*** parameter points to an array of WEB_STORAGE_OBJECT structures, this parameter must be initialized to the maximum size of the array being passed to the function. If the ***lpObjects*** parameter is NULL, this value must be initialized to zero, and when the function returns it will contain the number of matching objects.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebEnumObjects** function can be used to enumerate the number of objects that match a given label, content type or both. If a content type is specified, it must be a valid MIME media content type designated using the type/subtype nomenclature. For example, "text/plain" or "image/jpeg". An invalid MIME type will cause the function to fail.

This function can be used in two ways. If the ***lpObjects*** parameter is NULL and the value passed by reference in the ***lpdwObjects*** parameter is initialized to zero, the function will return the number of matching objects in ***lpdwObjects***. This can be used to dynamically determine the size of the lpObjects array instead of declaring a fixed-size array in your application.

If the ***lpObjects*** parameter is not NULL, then the value referenced by ***lpdwObjects*** must be initialized to the maximum size of the array that ***lpObjects*** points to. It is important to note that if either parameter is not initialized correctly, it can result in memory corruption and/or an unhandled exception.

The **WebGetFirstObject** and **WebGetNextObject** functions can be used to iterate through all matching storage objects without requiring you to preallocate memory for an array. Using **WebGetFirstObject** and **WebGetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

Example

```
DWORD dwObjects = 0;

if (WebEnumObjects(hStorage, _T("*.pdf"), NULL, 0, NULL, &dwObjects))
{
    LPWEB_STORAGE_OBJECT lpObjects = NULL;
    DWORD dwIndex;

    // Return if there are no matching objects in the container
    if (dwObjects == 0)
        return;

    // Allocate memory for an array of WEB_STORAGE_OBJECT structures
    lpObjects = (LPWEB_STORAGE_OBJECT)LocalAlloc(LPTR, sizeof(WEB_STORAGE_OBJECT) *
dwObjects);

    if (lpObjects == NULL)
        return; // Exhausted virtual memory?

    // Enumerate the matching objects
    if (!WebEnumObjects(hStorage, NULL, NULL, 0, lpObjects, &dwObjects))
        return;

    // Print information about each object
    for (dwIndex = 0; dwIndex < dwObjects; dwIndex++)
    {
        _tprintf(_T("Object: %s\n"), lpObjects[dwIndex].szObjectId);
        _tprintf(_T("Label: %s\n"), lpObjects[dwIndex].szLabel);
        _tprintf(_T("Size: %lu\n"), lpObjects[dwIndex].dwObjectSize);
        _tprintf(_T("Digest: %s\n"), lpObjects[dwIndex].szDigest);
        _tprintf(_T("Content: %s\n"), lpObjects[dwIndex].szContent);

        if (dwObjects > 1 && dwIndex < dwObjects - 1)
            _tprintf(_T("\n"));
    }

    LocalFree((HLOCAL)lpObjects);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetFirstObject, WebGetNextObject, WebGetObjectInformation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebEventProc Function

```
VOID CALLBACK WebEventProc(
   HSTORAGE hStorage,
   UINT nEventId,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **WebEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hStorage

A handle to the storage container.

nEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **WebRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **WebRegisterEvent** function. The **WebEventProc** function is a placeholder for the application-defined function name.

To obtain information about the current status of the read or write operation, call the **WebGetTransferStatus** function. An event handler can cancel the current operation by calling the **WebCancelTransfer** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCancelTransfer, WebGetTransferStatus, WebRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetAccountId Function

```
BOOL WINAPI WebGetAccountId(
   LPCTSTR lpszAccountId,
   INT nMaxLength
);
```

The **WebGetAccountId** function returns the web services account ID associated with the current session.

Parameters

lpszAccountId

A null-terminated string that will contain the account ID when the function returns. This parameter cannot be NULL and must be at least 35 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszAccountId* string parameter, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The account ID is a string that uniquely identifies the web services account that is associated with the session. The account ID corresponds with your product serial number and runtime license key, but it is not identical to either of those values.

If you are using an evaluation license, the account ID is temporary and only valid during the evaluation period. After the evaluation period has expired, the account ID is revoked and objects stored using this ID will be deleted. It is not recommended that you store critical application data using an evaluation license.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetStorageId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetErrorString Function

```
INT WINAPI WebGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **WebGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **WebGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetLastError, WebSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetFile Function

```
BOOL WINAPI WebGetFile(
   HSTORAGE hStorage,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel
);
```

The **WebGetFile** function downloads the contents of a storage object and copies it to a local file.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFile** function downloads the contents of the storage object and stores it in a local file. For information about the stored object that has been downloaded, use the **WebGetFileEx** function.

If you are downloading a large object and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFileEx, WebGetObject, WebPutFile, WebPutObject, WebRegisterEvent, WebUploadFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetFileEx Function

```
BOOL WINAPI WebGetFileEx(
   HSTORAGE hStorage,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   DWORD dwReserved,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebGetFileEx** function downloads the contents of a storage object and copies it to a local file.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFileEx** function downloads the contents of the storage object and stores it in a local file. Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are downloading a large object and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

The **WebDownloadFile** and **WebDownloadFileEx** functions provide similar functionality, but do not require a handle to an open storage container.

Example

```
WEB_STORAGE_OBJECT webObject;

// Download the object and store the contents in a local file
if (WebGetFileEx(hStorage, lpszLocalFile, lpszObjectLabel, 0, &webObject))
```

```
{
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %I64u\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be retrieved, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to retrieve \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFile, WebGetObjectEx, WebPutFileEx, WebPutObjectEx, WebRegisterEvent, WebUploadFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetLastError Function

```
DWORD WINAPI WebGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **WebSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **WebGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **WebSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WEB_ERROR. Those functions which call **WebSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetErrorString, WebSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetFirstApplication Function

```
BOOL WINAPI WebGetFirstApplication(
   LPWEB_STORAGE_APPLICATION lpAppInfo,
   LPDWORD lpdwContext,
);
```

The **WebGetFirstApplication** function returns information about the first registered application for the current storage account.

Parameters

lpAppInfo

A pointer to a WEB_STORAGE_APPLICATION structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

lpdwContext

A pointer to an unsigned integer that is used with subsequent calls to the **WebGetNextApplication** function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFirstApplication** function returns information about the first registered application. It is used in conjunction with the **WebGetNextApplication** function to enumerate all of the registered application IDs associated with your storage account.

This function provides an alternative to the **WebEnumApplications** function, which populates an array of WEB_STORAGE_APPLICATION structures. In some cases, iterating through each AppID in a loop may be preferred to preallocating memory for an array to store every registered application. Using **WebGetFirstApplication** and **WebGetNextApplication** is more efficient when you have registered a large number of application IDs.

You cannot intermix calls between **WebGetFirstApplication** and **WebEnumApplications**. The **WebEnumApplications** function will reset the internal application ID cache for the client session and subsequent calls to **WebGetNextApplication** will fail. The cached application ID information used by this function is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multithreaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;
DWORD dwContext = 0;

if (WebGetFirstApplication(&webApp, &dwContext))
{
    do
    {
        // Print information for each registered application
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);
```

```
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);
        _tprintf(_T("\n"));
    }
    while (WebGetNextApplication(&webApp, &dwContext));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebEnumApplications, WebGetNextApplication, WebRegisterAppId, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetFirstObject Function

```
BOOL WINAPI WebGetFirstObject(
   HSTORAGE hStorage,
   LPCTSTR lpszMatchLabel,
   LPCTSTR lpszContentType,
   DWORD dwReserved,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebGetFirstObject** function returns information about the first storage object that matches the specified label or content type.

Parameters

hStorage

A handle to the storage container.

lpszMatchLabel

A null-terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this value is a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A null-terminated string which specifies the content type of the objects to be enumerated. If this value is a NULL pointer or an empty string, the content type is ignored and all matching objects are returned. If a content type is specified, it must be a valid MIME media content type designated using the **type/subtype** nomenclature.

dwReserved

An unsigned integer value that is reserved for future use. This value must be zero.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFirstObject** function returns information about the first object that matches a given label, content type or both. It is used in conjunction with the **WebGetNextObject** function to enumerate all of the matching objects in the storage container.

This function provides an alternative to the **WebEnumObjects** function, which populates an array of WEB_STORAGE_OBJECT structures. In some cases, iterating over each object in a loop may be preferred to preallocating memory for an array to store every matching object. Using **WebGetFirstObject** and **WebGetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

You cannot intermix calls between **WebGetFirstObject** and **WebEnumObjects**. The

WebEnumObjects function will reset the internal object cache for the client session and subsequent calls to **WebGetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;

if (WebGetFirstObject(hStorage, _T("*.pdf"), NULL, 0, &webObject))
{
    do
    {
        // Print information about each object
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);
        _tprintf(_T("Label: %s\n"), webObject.szLabel);
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);
        _tprintf(_T("Content: %s\n"), webObject.szContent);
        _tprintf(_T("\n"));
    }
    while (WebGetNextObject(hStorage, &webObject));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebEnumObjects, WebGetNextObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetLocation Function

```
BOOL WINAPI WebGetLocation(
   LPWEB_LOCATION lpLocation
);
```

The **WebGetLocation** function returns the current physical location of the local computer system.

Parameters

lpLocation

A pointer to a WEB_LOCATION structure that will contain information about the location when the function returns. The structure members will be automatically initialized to zero and zero-length strings when the function is called. This parameter cannot be a NULL pointer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetLocation** function is used to obtain physical location information associated with the with the external IP address of the local system. The accuracy of this information can vary depending on the location, with the most detailed information being available for North America. The country and time zone information for all locations is generally accurate. However, as the location information becomes more precise, details such as city names, postal codes and specific geographic locations (e.g.: longitude and latitude) may have reduced accuracy.

This location information should not be used by programs that require extremely accurate map coordinates, such as navigation applications. The location information in North America should be generally accurate within a 25 mile (40km) radius. However, given the nature of how IP address location works, there is no guarantee that location information for any specific IP address or network will be accurate.

Software that is designed to protect the privacy of users, such as those which route all Internet traffic through proxy servers or VPNs, can significantly impact the accuracy of this information. In this case, the data returned in this structure may reflect the location of the network or proxy server, and not the location of the person using your application. It is recommended that you always request permission from the user before acquiring their location, have them confirm that the location is correct and provide a mechanism for them to update that information.

Example

```
WEB_LOCATION webLocation;

// Return information about the location of the local system
if (WebGetLocation(&webLocation))
{
    _tprintf(_T("Location ID: %s\n"), webLocation.szLocationId);
    _tprintf(_T("IP address: %s\n"), webLocation.szIPAddress);
    _tprintf(_T("ASN: %u\n"), webLocation.nAutonomousSystemNumber);
    _tprintf(_T("Organization: %s\n"), webLocation.szOrganization);
    _tprintf(_T("Region Name: %s\n"), webLocation.szRegionName);
    _tprintf(_T("Region Code: %03u\n"), webLocation.nRegionCode);
    _tprintf(_T("Country Name: %s\n"), webLocation.szCountryName);
    _tprintf(_T("Country Code: %03u\n"), webLocation.nCountryCode);
```

```
    _tprintf(_T("Subdivision: %s\n"), webLocation.szSubdivision);
    _tprintf(_T("City Name: %s\n"), webLocation.szCityName);
    _tprintf(_T("Postal Code: %s\n"), webLocation.szPostalCode);
    _tprintf(_T("Timezone: %s (%ld)\n"), webLocation.szTimezone,
webLocation.nTimezoneOffset);
    _tprintf(_T("Local Time: %04d-%02d-%02d %02d:%02d:%02d %s\n"),
             webLocation.stLocalTime.wYear,
             webLocation.stLocalTime.wMonth,
             webLocation.stLocalTime.wDay,
             webLocation.stLocalTime.wHour,
             webLocation.stLocalTime.wMinute,
             webLocation.stLocalTime.wSecond,
             webLocation.szTzShortName);
    _tprintf(_T("Coordinates: %s\n"), webLocation.szCoordinates);
    _tprintf(_T("Latitude: %.4f\n"), webLocation.dLatitude);
    _tprintf(_T("Longitude: %.4f\n"), webLocation.dLongitude);
}
else
{
    _tprintf(_T("Unable to determine the current location\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetNextApplication Function

```
BOOL WINAPI WebGetNextApplication(
   LPWEB_STORAGE_APPLICATION lpAppInfo,
   LPDWORD lpdwContext,
);
```

The **WebGetNextApplication** function returns information about the next registered application for the current storage account.

Parameters

lpAppInfo

A pointer to a WEB_STORAGE_APPLICATION structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

lpdwContext

A pointer to an unsigned integer that is used with subsequent calls to the **WebGetNextApplication** function. The value of this parameter is initialized by calling the **WebGetFirstApplication** function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetNextApplication** function returns information about the next registered application after initially calling the **WebGetFirstApplication** function. It is used to enumerate all of the registered application IDs associated with your storage account.

This function provides an alternative to the **WebEnumApplications** function, which populates an array of WEB_STORAGE_APPLICATION structures. In some cases, iterating through each AppID in a loop may be preferred to preallocating memory for an array to store every registered application. Using **WebGetFirstApplication** and **WebGetNextApplication** is more efficient when you have registered a large number of application IDs.

You cannot intermix calls between **WebGetNextApplication** and **WebEnumApplications**. The **WebEnumApplications** function will reset the internal application ID cache for the client session and subsequent calls to **WebGetNextApplication** will fail. The cached application ID information used by this function is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multithreaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;
DWORD dwContext = 0;

if (WebGetFirstApplication(&webApp, &dwContext))
{
    do
    {
        // Print information for each registered application
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);
```

```
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);
        _tprintf(_T("\n"));
    }
    while (WebGetNextApplication(&webApp, &dwContext));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebEnumApplications, WebGetFirstApplication, WebRegisterAppId, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetNextObject Function

```
BOOL WINAPI WebGetNextObject(
   HSTORAGE hStorage,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebGetNextObject** function returns information about the next storage object that matches the specified label or content type.

Parameters

hStorage

A handle to the storage container.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetNextObject** function returns information about the next object that matches the label and/or content type that was specified by the **WebGetFirstObject** function call. This function may only be called after **WebGetFirstObject** has been called, otherwise it will fail.

When all matching objects have been returned to the caller, this function will return zero (FALSE) and the **WebGetLastError** function will return NO_ERROR. Any other error code indicates an underlying problem with the request, such as an invalid parameter passed to the function.

You cannot intermix calls between **WebGetNextObject** and **WebEnumObjects**. The **WebEnumObjects** function will reset the internal object cache for the client session and subsequent calls to **WebGetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;

if (WebGetFirstObject(hStorage, _T("*.pdf"), NULL, 0, &webObject))
{
    do
    {
        // Print information about each object
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);
        _tprintf(_T("Label: %s\n"), webObject.szLabel);
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);
        _tprintf(_T("Content: %s\n"), webObject.szContent);
        _tprintf(_T("\n"));
    }
    while (WebGetNextObject(hStorage, &webObject));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebEnumObjects, WebGetFirstObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetObject Function

```
BOOL WINAPI WebGetObject(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPVOID lpvBuffer,
   DWORD lpdwLength
);
```

The **WebGetObject** function retrieves the contents of a storage object and copies it to the memory buffer that is provided.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObject** function is used to retrieve a stored object from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the object. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated and the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD dwLength = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
```

```
// returned in the hgblBuffer parameter
if (WebGetObject(hStorage, lpszObjectLabel, &hgblBuffer, &dwLength))
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetFile, WebGetObjectEx, WebPutFile, WebPutObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetObjectEx Function

```
BOOL WINAPI WebGetObjectEx(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   DWORD dwReserved,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   LPWEB_STORAGE_OBJECT lpObject,
);
```

The **WebGetObjectEx** function retrieves the contents of a storage object and copies it to the memory buffer that is provided. Additional information about the object is returned to the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

dwReserved

An unsigned integer that is reserved for future use. This value should always be zero.

lpvBuffer

A buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the ***lpvBuffer*** parameter. If the ***lpvBuffer*** parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectEx** function is used to retrieve a stored object from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the object. In this case, the ***lpvBuffer*** parameter will point to the buffer that was allocated and the value that the ***lpdwLength*** parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the ***lpvBuffer*** parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the ***lpdwLength*** parameter points to must be initialized to zero. It is important to note that the memory

handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

If you wish to retrieve the contents of a text object and store it in a null-terminated string buffer, use the **WebGetTextObject** function. If you wish to download the contents of a stored file, use the **WebGetFileEx** function.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD dwLength = 0;
WEB_STORAGE_OBJECT webObject;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
if (WebGetObjectEx(hStorage, lpszObjectLabel, 0, &hgblBuffer, &dwLength,
&webObject))
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetFileEx, WebGetObject, WebGetTextObject, WebPutFileEx, WebPutObjectEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetObjectInformation Function

```
BOOL WINAPI WebGetObjectInformation(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   DWORD dwReserved,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebGetObjectInformation** function retrieves the metadata for a storage object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object.

dwReserved

An unsigned integer that is reserved for future use. This value should always be zero.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter cannot be a NULL pointer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectInformation** function is used to retrieve the metadata for a stored object on the server, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents. This function can also be used as a simple method to determine if the specified object exists without the overhead of requesting the server attempt to retrieve the contents of an object.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

If you are only interested in obtaining the size of a stored object, you can use the **WebGetObjectSize** function.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **WebGetStorageQuota** function.

Example

```
WEB_STORAGE_OBJECT webObject;

// Get information about the object
if (WebGetObjectInformation(hStorage, lpszObjectLabel, 0, &webObject))
{
    // The object exists, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
```

```
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object does not exist, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to get information on \"%s\" (%s)\n"), lpszObjectLabel,
szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetFile, WebGetObject, WebGetObjectSize, WebGetObjectTime, WebGetStorageQuota, WebPutFile, WebPutObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetObjectSize Function

```
BOOL WINAPI WebGetObjectSize(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPDWORD lpdwObjectSize
);
```

The **WebGetObjectSize** function returns the size of the stored object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object. This parameter cannot be a NULL pointer.

lpdwObjectSize

A pointer to an unsigned integer value that will contain the size of the object. If this parameter is NULL, the parameter is ignored and the function only checks for the existence of an object that matches the specified label.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectSize** function is used to retrieve the size of a stored object on the server. This function can also be used as a simple method to determine if the specified object exists. An object size of zero means the object exists, but currently has no data associated with it.

If the *lpObjectSize* parameter is not NULL, its value is initialized to zero when the function is called, and updated with the object size when the function returns.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting the size of an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

The **WebGetObjectInformation** function can be used to obtain the metadata associated with the storage object, including the size, content type, creation date and a SHA-256 digest of the data.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **WebGetStorageQuota** function.

Example

```
// Check if the object exists
DWORD dwObjectSize;

if (WebGetObjectSize(hStorage, lpszObjectLabel, &dwObjectSize))
{
    // The object exists, display the size in bytes
    _tprintf(_T("The size of \"%s\" is %lu bytes\n"), lpszObjectLabel,
dwObjectSize);
}
```

```
else
{
    // The object size could not be determined, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to get the size of \"%s\" (%s)\n"), lpszObjectLabel,
szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetObject, WebGetObjectInformation, WebGetObjectTime, WebGetStorageQuota, WebPutObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetObjectTime Function

```
BOOL WINAPI WebGetObjectTime(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPFILETIME lpTimeCreated,
   LPFILETIME lpTimeModified
);
```

The **WebGetObjectTime** function returns the creation and last modification time of the stored object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object. This parameter cannot be a NULL pointer.

lpTimeCreated

A pointer to a FILETIME structure which will contain the time the storage object was created when the function returns. If this information is not required, the parameter may be NULL.

lpTimeModified

A pointer to a FILETIME structure which will contain the time the storage object was last modified when the function returns. If this information is not required, the parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectTime** function is used to retrieve the creation and modification times for a stored object on the server. This function can also be used as a simple method to determine if the specified object exists. Each of the LPFILETIME parameters are optional and may be NULL if either the creation or modification times are not needed. If a pointer to a FILETIME structure is provided, its members will be initialized to zero and then updated with the creation and/or modification time if the function succeeds.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting the size of an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

If you wish to obtain all of the metadata associated with a specific object with a single function call, including the size, content type, creation and modification times, use the **WebGetObjectInformation** function.

Example

```
// Check if the object exists
FILETIME ftCreated;
FILETIME ftModified;

if (WebGetObjectTime(hStorage, lpszObjectLabel, &ftCreated, &ftModified))
```

```
{
    // The object exists, display the last modification time
    TCHAR szLocalDate[128];
    TCHAR szLocalTime[128];
    FILETIME ftLocalTime;
    SYSTEMTIME stLocalTime;

    FileTimeToLocalFileTime(&ftModified, &ftLocalTime);
    FileTimeToSystemTime(&ftLocalTime, &stLocalTime);

    GetDateFormat(LOCALE_USER_DEFAULT,
                  DATE_SHORTDATE,
                  &stLocalTime,
                  NULL,
                  szLocalDate, 128);

    GetTimeFormat(LOCALE_USER_DEFAULT,
                  TIME_FORCE24HOURFORMAT,
                  &stLocalTime,
                  NULL,
                  szLocalTime, 128);

    _tprintf(_T("The object \"%s\" was lasted modified on %s %s\n"),
lpszObjectLabel, szLocalDate, szLocalTime);
}
else
{
    // The object times could not be determined, display an error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to get the size of \"%s\" (%s)\n"), lpszObjectLabel,
szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetObject, WebGetObjectInformation, WebGetStorageQuota, WebPutObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetPosixTime32 Function

```
BOOL WINAPI WebGetPosixTime32(
    LONG *lpTime32
);
```

The **WebGetPosixTime32** function returns a 32-bit POSIX timestamp value.

Parameters

lpTime32

A pointer to a signed 32-bit integer which will contain a POSIX timestamp value when the function returns. This value will be initialized to zero when the function is called and then updated when it returns successfully. This parameter cannot be NULL or the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetPosixTime32** function returns the current date and time as a 32-bit POSIX timestamp, expressed as the number of whole seconds that have elapsed since 00:00:00 UTC on January 1, 1970. The value returned by this function is based on Coordinated Universal Time (UTC) and is not affected by the local time zone or daylight saving time settings.

The timestamp returned by this function is obtained from the SocketTools API server and may be cached locally using the same internal time synchronization mechanisms used by other functions in this library. This helps ensure that the value reflects an accurate UTC time while avoiding unnecessary network time queries when synchronization is performed periodically.

This function is provided for compatibility with legacy applications or development environments that cannot work with 64-bit integer values. Whenever possible, applications should use **WebGetPosixTime64** instead. A 64-bit POSIX timestamp avoids the limitations of 32-bit values and should be preferred for all new development.

A 32-bit POSIX timestamp can only represent dates up to January 19, 2038. This limitation is commonly referred to as the Year 2038 problem. Unlike some implementations which allow the value to overflow and wrap around, this function will fail if the timestamp exceeds the maximum value that can be represented by a signed 32-bit integer. This means that after that date, this function will no longer return a valid timestamp.

This function returns the timestamp as a signed 32-bit integer for compatibility with common C and C++ data types. Under normal conditions the value returned by this function will always be a positive number representing a date after the POSIX epoch. The timestamp is returned with one-second precision; fractional seconds are not included.

If the function cannot obtain the current universal time or the time value cannot be converted to a valid POSIX timestamp, the function will fail and the value pointed to by *lpTime32* will remain zero.

Example

```
LONG posixTime32;

if (WebGetPosixTime32(&posixTime32))
{
```

```
    // Display the current POSIX timestamp value
    _tprintf(_T("The current POSIX timestamp value is %ld\n"), posixTime32);
}
else
{
  DWORD dwError = WebGetLastError();
  _tprintf(_T("Unable to obtain a POSIX timestamp value, error 0x%08lx\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetPosixTime64, WebGetSystemTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetPosixTime64 Function

```
BOOL WINAPI WebGetPosixTime64(
    LONGLONG *lpTime64
);
```

The **WebGetPosixTime64** function returns a 64-bit POSIX timestamp value.

Parameters

lpTime64

A pointer to a signed 64-bit integer which will contain a POSIX timestamp value when the function returns. This value will be initialized to zero when the function is called and then updated when it returns successfully. This parameter cannot be NULL or the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetPosixTime64** function returns the current date and time as a 64-bit POSIX timestamp, expressed as the number of whole seconds that have elapsed since 00:00:00 UTC on January 1, 1970. The value returned by this function is based on Coordinated Universal Time (UTC) and is not affected by the local time zone or daylight saving time settings.

The timestamp returned by this function is obtained from the SocketTools API server and may be cached locally using the same internal time synchronization mechanisms used by other functions in this library. This helps ensure that the value reflects an accurate UTC time while avoiding unnecessary network time queries when synchronization is performed periodically.

Whenever possible, applications should use this function instead of **WebGetPosixTime32**. A 32-bit POSIX timestamp can only represent dates up to January 19, 2038, after which it overflows and can no longer represent later dates correctly. This limitation is commonly referred to as the Year 2038 problem. Because this function returns a 64-bit value, it avoids that limitation and should be preferred for all new development.

This function returns the timestamp as a signed 64-bit integer for compatibility with common C and C++ data types. Under normal conditions the value returned by this function will always be a positive number representing a date after the POSIX epoch. The timestamp is returned with one-second precision; fractional seconds are not included.

If the function cannot obtain the current universal time or the time value cannot be converted to a valid POSIX timestamp, the function will fail and the value pointed to by *lpTime64* will remain zero.

Example

```
LONGLONG posixTime64;

if (WebGetPosixTime64(&posixTime64))
{
    // Display the current POSIX timestamp value
    _tprintf(_T("The current POSIX timestamp value is %I64d\n"), posixTime64);
}
else
{
```

```
  DWORD dwError = WebGetLastError();
  _tprintf(_T("Unable to obtain a POSIX timestamp value, error 0x%08lx\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetPosixTime32, WebGetSystemTime

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetStorageId Function

```
BOOL WINAPI WebGetStorageId(
   HSTORAGE hStorage,
   LPCTSTR lpszStorageId,
   INT nMaxLength
);
```

The **WebGetStorageId** function returns the storage container ID.

Parameters

hStorage

A handle to the storage container.

lpszStorageId

A null-terminated string that will contain the storage ID when the function returns. This parameter cannot be NULL and must be at least 35 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the ***lpszStorageId*** string parameter, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The storage ID is a string that identifies the storage container that was opened with the **WebOpenStorage** function. The storage ID is associated with your development license and is guaranteed to be a unique value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebGetAccountId, WebOpenStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetStorageQuota Function

```
BOOL WINAPI WebGetStorageQuota(
   LPWEB_STORAGE_QUOTA lpQuota
);
```

The **WebGetStorageQuota** function returns quota limits assigned to your storage account.

Parameters

lpQuota

A pointer to a WEB_STORAGE_QUOTA structure that will contain information the quota limits for your account when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetStorageQuota** function can be used to determine how much storage space is available to your application. The *ulBytesFree* structure member will tell you how many bytes of storage you have available, and the *dwObjectSize* member will tell you the maximum size of any individual storage object. In addition to the bytes allocated for storage, there is also a limit on the total number of objects that your application may create which is specified by the *dwObjectLimit* member.

Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

Example

```
WEB_STORAGE_QUOTA appQuota;

// Initialize the library
if (WebInitialize(CSTOOLS12_LICENSE_KEY, NULL) == FALSE)
{
    _tprintf(_T("Unable to initialize the SocketTools library\n"));
    return;
}

// Display storage account usage and limits
if (WebGetStorageQuota(&appQuota))
{
     _tprintf(_T("Objects Used:  %lu\n"), appQuota.dwObjects);
     _tprintf(_T("Object Limit:  %lu\n"), appQuota.dwObjectLimit);
     _tprintf(_T("Object Size:   %lu\n"), appQuota.dwObjectSize);
     _tprintf(_T("Bytes Used:    %I64u\n"), appQuota.ulBytesUsed);
     _tprintf(_T("Bytes Free:    %I64u\n"), appQuota.ulBytesFree);
     _tprintf(_T("Storage Limit: %I64u\n"), appQuota.ulStorageLimit);
}
else
{
    // Unable to get the quota information for this account
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
```

```
    _tprintf(_T("Unable to get the account quota (%s)\n"), szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetObjectInformation, WebGetObjectSize, WEB_STORAGE_QUOTA

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetStorageTimeout Function

```
DWORD WINAPI WebGetStorageTimeout(
    HSTORAGE hStorage
);
```

The **WebGetStorageTimeout** function returns the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The default timeout period is 10 seconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebOpenStorage, WebSetStorageTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetSystemTime Function

```
BOOL WINAPI WebGetSystemTime(
   DWORD dwOptions,
   LPSYSTEMTIME lpDateTime
);
```

The **WebGetSystemTime** function returns the current date and time in a SYSTEMTIME structure.

Parameters

dwOptions

An integer parameter which specifies one or more options.

Value	Description
WEB_TIME_DEFAULT	The default options should be used when retrieving the date and time. This is the same as specifying WEB_TIME_UNIVERSAL.
WEB_TIME_UNIVERSAL	The date and time values returned in the SYSTEMTIME structure will use Univeral Coordinated Time (UTC) and will not be adjusted for the current timezone. This is similar to the value returned by the **GetSystemTime** function in the Win32 API.
WEB_TIME_LOCAL	The date and time values returned in the SYSTEMTIME structure will adjusted for the current timezone. This is similar to the value returned by the **GetLocalTime** function in the Win32 API. The local timezone is determined by system settings for the current user.

lpDateTime

A pointer to a SYSTEMTIME structure which contains the current date and time. The structure members will be initialized to zero when the function is called and then updated when it returns successfully. This parameter cannot be NULL or the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetSystemTime** function returns the current date and time into a SYSTEMTIME structure. Unlike the **GetSystemTime** function, this function does not use the local system clock. Instead, it obtains the current time from the SocketTools API server, which is synchronized with official time servers using the Network Time Protocol (NTP). This helps ensure that the returned value is accurate and has not been altered by the user or affected by incorrect system clock settings.

This function can be useful in scenarios where an application must rely on a trusted and consistent time source. For example, it may be used for validating software licenses, enforcing time-based restrictions, generating timestamps for logging or auditing, or coordinating events across multiple systems where the local system time cannot be assumed to be accurate.

Because the time value is retrieved from a remote server, this function requires an active network connection and may be subject to network latency or temporary connectivity issues. To reduce network overhead, the function internally caches the most recently retrieved time value for a short

period (typically up to 60 seconds). When the function is called again within this interval, it does not immediately query the server. Instead, it calculates the elapsed time since the last successful request using a high-resolution system tick count and adjusts the cached time value accordingly.

This approach allows the returned time to continue advancing between server synchronizations, providing a smooth and consistent progression of time values without requiring a network request for each call. However, because this adjustment is based on the local system timer, it should be considered an approximation of the actual server time. The time value will be re-synchronized with the server periodically to maintain overall accuracy.

In most cases, it is recommended that applications use the **WEB_TIME_UNIVERSAL** option when storing or comparing date/time values. Universal (UTC) time provides a consistent, timezone-independent reference that avoids issues related to daylight saving time changes or differences in local system settings. This is particularly important when timestamps are shared between systems or persisted for later comparison.

The **WEB_TIME_LOCAL** option can be used when the application requires a date/time value adjusted for the user's current timezone, such as when displaying timestamps in a user interface. However, because the timezone adjustment is based on the local system configuration, it may still be affected by incorrect timezone or daylight saving settings on the client system.

Developers should consider using the local system time (for example, via **GetSystemTime** or **GetLocalTime**) when absolute accuracy is not critical or when network access is unavailable. Retrieving the time locally avoids the overhead of a network request and is typically sufficient for general-purpose tasks such as UI updates, file timestamps, or short-lived timing operations.

This function is not intended to replace standard system time APIs for high-frequency or real-time operations. Applications that require precise time measurements or guaranteed monotonic behavior should use dedicated high-resolution timing APIs instead of relying on this function.

Example

```
SYSTEMTIME stDateTime;

if (WebGetSystemTime(WEB_TIME_UNIVERSAL, &stDateTime))
{
    // Display the current date and time in ISO 8601 format
    _tprintf(_T("The current date and time is %04hu-%02hu-
%02huT%02hu:%02hu:%02huZ\n"),
             stDateTime.wYear,
             stDateTime.wMonth,
             stDateTime.wDay,
             stDateTime.wHour,
             stDateTime.wMinute,
             stDateTime.wSecond);
}
else
{
  DWORD dwError = WebGetLastError();
  _tprintf(_T("Unable to determine the current date and time, error 0x%08lx\n"),
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cswebv12.lib

See Also

WebGetPosixTime32, WebGetPosixTime64, SYSTEMTIME

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetTextObject Function

```
BOOL WINAPI WebGetTextObject(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPTSTR lpszBuffer,
   INT nMaxLength,
   LPWEB_STORAGE_OBJECT lpObject,
);
```

The **WebGetTextObject** function retrieves the contents of a text storage object and copies it to the string buffer that is provided. Additional information about the object is returned to the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the text object that should be retrieved from the server.

lpszBuffer

A null-terminated string buffer which will contain the text stored in the object. The string must be large enough to store the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be large enough to store the complete text and the terminating null character.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetTextObject** function is used to retrieve the text from a stored object and copy it into a string buffer. The string buffer must be pre-allocated and large enough to contain the entire text and a terminating null character.

If you use the **WebGetObjectSize** function to determine the size of the object so your application can dynamically allocate memory for the string buffer, be sure to allocate a string buffer that is larger than the reported size. The size of the stored text object may differ slightly from what is returned in the string, depending on how the text has been encoded.

Text objects that are created or updated using the Unicode version of **WebPutTextObject** function will always be converted and stored as UTF-8 encoded text. If your application uses the ANSI version of this function to retrieve that text, it will be UTF-8 encoded and your application will need to convert it back to UTF-16 encoded text, if needed. The Unicode version of this function will perform the conversion automatically.

This function should never be used to retrieve data from an object that does not contain text. If the **WebPutObject** function was previously used to store UTF-16 encoded text, you must use **WebGetObject** to retrieve that data. This function only recognizes UTF-8 encoded text and considers UTF-16 encoded text to be binary data.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
LPTSTR lpszBuffer = (LPTSTR)NULL;
INT nMaxLength = 8192;
WEB_STORAGE_OBJECT webObject;

lpszBuffer = (LPTSTR)LocalAlloc(LPTR, nMaxLength);
if (lpszBuffer == NULL)
{
    // Unable to allocate memory for the string
    _ftprintf(stderr, _T("Unable to allocate %d bytes of memory\n"), nMaxLength);
    return;
}

if (WebGetTextObject(hStorage, lpszObjectLabel, lpszBuffer, nMaxLength,
&webObject))
{
    // Output the text contained in the object
    _fputts(lpszBuffer, stdout);
}
else
{
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _ftprintf(stderr, _T("Unable to get \"%s\": %s\n"), lpszObjectLabel, szError);
}

// Free memory allocated for the text when no longer needed
LocalFree((HLOCAL)lpszBuffer);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCompareText, WebGetFileEx, WebGetObject, WebPutFileEx, WebPutObjectEx, WebPutTextObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebGetTransferStatus Function

```
BOOL WINAPI WebGetTransferStatus(
   HSTORAGE hStorage,
   LPWEB_STORAGE_TRANSFER lpStatus
);
```

The **WebGetTransferStatus** function returns information about the current data transfer in progress.

Parameters

hStorage

Handle to the storage container.

lpStatus

A pointer to a WEB_STORAGE_TRANSFER structure which contains information about the status of the current data transfer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetTransferStatus** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this function will return the status of the last successful data transfer.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified storage session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCancelTransfer, WebRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebInitialize Function

```
BOOL WINAPI WebInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **WebInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **WebInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **WebUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **WebInitialize** or **WebUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCloseStorage, WebOpenStorage, WebRegisterAppId, WebUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebMoveObject Function

```
BOOL WINAPI WebMoveObject(
  HSTORAGE hStorage,
  LPCTSTR lpszOldLabel,
  LPCTSTR lpszNewLabel,
  DWORD dwStorageType,
  LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebMoveObject** function moves an existing object to a different storage container.

Parameters

hStorage

A handle to the storage container.

lpszOldLabel

A null-terminated string which specifies the name of the existing storage object to be moved. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A null-terminated string which specifies a new label for the storage object being moved. This parameter may be NULL or point to an empty string, in which case the label name is not changed.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local

	machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain information about the new storage object. If this information is not required, this parameter may be a NULL pointer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebMoveObject** function is used to move an existing storage object to a new container. For example, it can move an object originally created using WEB_STORAGE_USER to the WEB_STORAGE_MACHINE container. The ***dwStorageType*** parameter must specify a valid storage container type and cannot be WEB_STORAGE_DEFAULT.

If the ***dwStorageType*** parameter specifies the same container that the object is currently in, and the ***lpszNewLabel*** parameter specifies a new label name, this function will simply rename the existing object and is effectively the same as calling the **WebRenameObject** function.

To duplicate an existing storage object, use the **WebCopyObject** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCopyObject, WebDeleteObject, WebRenameObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebOpenStorage Function

```
HSTORAGE WINAPI WebOpenStorage(
    LPCTSTR lpszAppId,
    DWORD dwStorageType
```

The **WebOpenStorage** function establishes a connection with the server and opens the storage container associated with the specified storage type.

Parameters

lpszAppId

A null-terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

Return Value

If the function succeeds, the return value is a handle to the storage container. If the function fails, the

return value is INVALID_HANDLE. To get extended error information, call **WebGetLastError**.

Remarks

The **WebOpenStorage** function opens the specified storage container and requests an access token for the application. This is the first function that must be called prior to accessing any stored objects.

The application ID is a string that uniquely identifies the application requesting the access and must have been previously registered with the server by calling the **WebRegisterAppId** function. If the *lpszAppId* parameter is NULL or an empty string, the function will use a default internal ID that is allocated for each storage account. You can use this default ID if you wish to share data between all of the applications you create.

The storage type specifies the type of container that objects will be stored in. In most cases, we recommend using WEB_STORAGE_GLOBAL which means that stored objects will be accessible to all users of your application. However, you can limit access to the stored objects based on the local domain, local machine ID or the current user SID.

If you specify anything other than global storage, objects can be orphaned if the system configuration changes. For example, if WEB_STORAGE_MACHINE is specified, the objects that are stored there can only be accessed from that computer system. If the system is reconfigured (for example, the boot volume formatted and Windows is reinstalled) the unique identifier for that system will change and the previous objects that were stored by your application can no longer be accessed.

It is advisable is to store critical application data and configuration information using WEB_STORAGE_GLOBAL and use other non-global storage containers for configuration information that is unique to that system and/or user which is not critical and can be easily recreated.

Example

```
 HSTORAGE hStorage = INVALID_HANDLE;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };
BOOL bSuccess = FALSE;

// Register the application ID which identifies the application
if (!WebRegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
hStorage = WebOpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);

if (hStorage == INVALID_HANDLE)
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}

// Upload a local file to the storage server and return information
// about the stored object when it completes
bSuccess = WebPutFileEx(hStorage,
                        lpszLocalFile,
                        lpszObjectLabel,
                        NULL,
```

```
                        WEB_OBJECT_DEFAULT,
                        0,
                        &webObject);

if (bSuccess)
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _tprintf(_T("WebPutFileEx failed, error 0x%08lX (%s)\n"), dwError, szError);
}

// Release the handle allocated for this storage session
WebCloseStorage(hStorage);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCloseStorage, WebGetFile, WebGetObject, WebGetStorageId, WebPutFile, WebPutObject, WebRegisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebPutFile Function

```
BOOL WINAPI WebPutFile(
   HSTORAGE hStorage,
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel
);
```

The **WebPutFile** function uploads the contents of a local file to a storage container.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be uploaded. If a path is not specified, the file will be read from the current working directory. The current user must have read access to the file, and an error will be returned if the function cannot obtain an exclusive lock on the file during the upload process.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutFile** function uploads the contents of a local file to the storage container specified by the *hStorage* handle. If an object with this name already exists, its data will be replaced with the contents of the file. The object will be created using the default attributes that permit read and write access, and will automatically determine the content type based on the type of file being uploaded.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

If you want to specify the object content type, its attributes, or obtain additional information about the storage object that was created, use the **WebPutFileEx** function.

The **WebValidateLabel** function can be used to ensure a label is valid prior to calling this function. Refer to that function for more information about the difference between Windows file names and object labels.

If you are uploading a large file and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Example

```
// Upload a local file to the storage container
if (WebPutFile(hStorage, lpszLocalFile, lpszObjectLabel)
{
```

```
    // The file was uploaded successfully
    _tprintf(_T("The file was stored as \"%s\"\n"), lpszObjectLabel);
}
else
{
    // The file could not be uploaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to store \"%s\" (%s)\n"), lpszLocalFile, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFile, WebPutFileEx, WebRegisterEvent, WebUploadFile, WebValidateLabel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebPutFileEx Function

```
BOOL WINAPI WebPutFileEx(
  HSTORAGE hStorage,
  LPCTSTR lpszLocalFile,
  LPCTSTR lpszObjectLabel,
  LPCTSTR lpszContentType,
  DWORD dwAttributes,
  DWORD dwReserved,
  LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebPutFileEx** function uploads the contents of a local file to a storage container and returns information about the new object.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be uploaded. If a path is not specified, the file will be read from the current working directory. The current user must have read access to the file, and an error will be returned if the function cannot obtain an exclusive lock on the file during the upload process.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

lpszContentType

A null-terminated string that identifies the contents of the file being uploaded. If this parameter is a NULL pointer, or specifies a zero-length string, the function will attempt to automatically determine the content type based on the file name extension and the contents of the file.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bit flags using a bitwise OR operation:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **WebEnumObjects** function. The

	object can only be accessed directly when specifying its label.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutFileEx** function uploads the contents of a local file to the storage container specified by the *hStorage* handle. The caller is responsible for specifying the object's content type and the attributes for the object being created.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text files have a content type of **text/plain** while an XML-formatted text file would have a content type of **text/xml**. Files that contain unstructured binary data are typically identified as **application/octet-stream**.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

The **WebValidateLabel** function can be used to ensure a label is valid prior to calling this function. Refer to that function for more information about the difference between Windows file names and object labels

If you are uploading a large file and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

The **WebUploadFile** and **WebUploadFileEx** functions provide similar functionality, but do not require a handle to an open storage container.

Example

```
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the storage container, automatically
// determining the content type with normal read/write access
if (WebPutFileEx(hStorage, lpszLocalFile, lpszObjectLabel, NULL, WEB_OBJECT_NORMAL,
0, &webObject))
{
    // The file was uploaded, display the object metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The file could not be uploaded, display the error
```

```
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to store \"%s\" (%s)\n"), lpszLocalFile, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFile, WebPutFile, WebRegisterEvent, WebUploadFile, WebValidateLabel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebPutObject Function

```
BOOL WINAPI WebPutObject(
  HSTORAGE hStorage,
  LPCTSTR lpszObjectLabel,
  LPCVOID lpvBuffer,
  DWORD dwLength
);
```

The **WebPutObject** function creates a new storage object or overwrites an existing object with the contents of the buffer provided.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

lpvBuffer

A buffer that contains the data to be stored. If this parameter is NULL, the ***dwLength*** parameter must have a value of zero and a storage object will be created which has no data associated with it.

dwLength

An unsigned integer value that specifies the number of bytes that will be copied from the ***lpvBuffer*** parameter and stored in the object. If the ***lpvBuffer*** parameter is NULL, this value must be zero or the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutObject** function stores the contents of a buffer to the storage container specified by the ***hStorage*** handle. If an object with this name already exists, its data will be replaced with the contents of the file. The object will be created using the default attributes that permit read and write access, and it will automatically determine the content type based on the data being stored.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created using this function. The object label cannot contain wildcard characters.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

If you want to specify the object content type, its attributes, or obtain additional information about the storage object that was created, use the **WebPutObjectEx** function.

If you want to upload the contents of a file, the **WebPutFile** function simplifies this process. The example code below demonstrates how the Windows API can be used to read from a local file and store the contents using **WebPutObject**.

If you are storing a large amount of data and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Example

```
HANDLE hFile = INVALID_HANDLE_VALUE;
LPBYTE lpContents = NULL;
DWORD dwLength = 0;
BOOL bFileRead = FALSE;

// Open a file on the local system and read the contents
// into a buffer that will be stored on the server
hFile = CreateFile(lpszLocalFile,
                   GENERIC_READ,
                   0,
                   NULL,
                   OPEN_EXISTING,
                   FILE_ATTRIBUTE_NORMAL,
                   NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
    // Unable to open the file
    return;
}

// Get the size of the file and allocate a buffer large
// enough to store the contents of the file
dwLength = GetFileSize(hFile, NULL);
lpContents = (LPBYTE)LocalAlloc(LPTR, dwLength + 1);

if (lpContents == NULL)
{
    // Memory allocation failed
    return;
}

bFileRead = ReadFile(hFile, lpContents, dwLength, &dwLength, NULL);
CloseHandle(hFile);

if (!bFileRead)
{
    // Unable to read the contents of the file
    return;
}

// Store the contents of the buffer
if (WebPutObject(hStorage, lpszObjectLabel, lpContents, dwLength))
{
    _tprintf(_T("WebPutObject stored %lu bytes\n"), dwLength);
}
else
{
    // The object could not be created
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to create \"%s\" (%s)\n"), lpszObjectLabel, szError);
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebPutFile, WebPutObjectEx, WebRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebPutObjectEx Function

```
BOOL WINAPI WebPutObjectEx(
  HSTORAGE hStorage,
  LPCTSTR lpszObjectLabel,
  LPCTSTR lpszContentType,
  DWORD dwAttributes,
  DWORD dwReserved,
  LPCVOID lpvBuffer,
  DWORD dwLength,
  LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebPutObjectEx** function stores the contents of buffer to a container and returns information about the new object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

lpszContentType

A null-terminated string that identifies the contents of the buffer being stored. If this parameter is a NULL pointer, or specifies a zero-length string, the function will attempt to automatically determine the content type based on the object label and the contents of the buffer.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bit flags using a bitwise OR operation:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **WebEnumObjects** function. The object can only be accessed directly when specifying its label.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

lpvBuffer

A buffer that contains the data to be stored. If this parameter is NULL, the ***dwLength*** parameter must have a value of zero and a storage object will be created which has no data associated with it.

dwLength

An unsigned integer value that specifies the number of bytes that will be copied from the ***lpvBuffer*** parameter and stored in the object. If the ***lpvBuffer*** parameter is NULL, this value must be zero or the function will fail.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that was created or replaced. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutObjectEx** function uploads the contents of a buffer to the storage container specified by the ***hStorage*** handle. The caller is responsible for specifying the object's content type and the attributes for the object being created.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created using this function. The object label cannot contain wildcard characters.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, typical text data has a content type of **text/plain** while XML-formatted text would have a content type of **text/xml**. Data which contains unstructured binary data is typically identified as **application/octet-stream**.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

If you wish to store the contents of a null-terminated string, you can use the **WebPutTextObject** function. If you want to upload the contents of a file, the **WebPutFileEx** function simplifies this process. The example code below demonstrates how the Windows API can be used to read from a local file and store the contents using **WebPutObjectEx**.

If you are storing a large amount of data and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Example

```
HANDLE hFile = INVALID_HANDLE_VALUE;
LPBYTE lpContents = NULL;
DWORD dwLength = 0;
BOOL bFileRead = FALSE;
WEB_STORAGE_OBJECT webObject;

// Open a file on the local system and read the contents
// into a buffer that will be stored on the server
hFile = CreateFile(lpszLocalFile,
                   GENERIC_READ,
```

```
                    0,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
    // Unable to open the file
    return;
}

// Get the size of the file and allocate a buffer large
// enough to store the contents of the file
dwLength = GetFileSize(hFile, NULL);
lpContents = (LPBYTE)LocalAlloc(LPTR, dwLength + 1);

if (lpContents == NULL)
{
    // Memory allocation failed
    return;
}

bFileRead = ReadFile(hFile, lpContents, dwLength, &dwLength, NULL);
CloseHandle(hFile);

if (!bFileRead)
{
    // Unable to read the contents of the file
    return;
}

// Store the contents of the buffer, identifying it as an unstructured
// stream of bytes, and the object is created with read/write access
if (WebPutObjectEx(hStorage,
                    lpszObjectLabel,
                    _T("application/octet-stream"),
                    WEB_OBJECT_NORMAL,
                    0,
                    lpContents,
                    dwLength,
                    &webObject))
{
    // The object was created or replaced, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be created
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to create \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebPutFileEx, WebPutObject, WebPutTextObject, WebRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebPutTextObject Function

```
BOOL WINAPI WebPutTextObject(
   HSTORAGE hStorage,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszObjectText,
   INT cchObjectText,
   LPWEB_STORAGE_OBJECT lpObject,
);
```

The **WebPutTextObject** function creates or replaces a text object with the contents of a string buffer. Additional information about the object is returned to the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A null-terminated string that specifies the label of the text object that should be created or updated.

lpszObjectText

A null-terminated string which contains the text that should be uploaded to the storage server. This parameter may be NULL, in which case the object is created but no data is stored.

cchObjectText

The number of characters that are stored in the ***lpszObjectText*** string. If this value is -1, the size of the string will be determined automatically by counting the number of characters up to the terminating null character. If the ***lpszObjectText*** parameter is NULL, this value must be zero.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutTextObject** function is used to store the contents of a string as a text object. The string should be terminated with a null character and cannot contain embedded nulls. If the Unicode version of this function is called, the contents of the string will be normalized prior to being converted to UTF-8 using canonical composition, where decomposed characters are combined to create their canonical precomposed equivalent.

The size of the text object that is created can be different from the length of the string buffer passed to this function, depending on how the text has been encoded and stored. This will almost always be the case when the Unicode version of this function is called because the UTF-16 string will be converted and stored as UTF-8 encoded text. This encoding will typically increase the size of the stored object unless the string only contains ASCII characters.

This function should only be used to store textual data in a null terminated string, and should never be used to store binary data. If you wish to create or update an object which contains binary data, use the **WebPutObjectEx** function instead. If you want to upload the contents of a file, the **WebPutFileEx**

function simplifies this process.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
LPCTSTR lpszObjectLabel = _T("LoremIpsum");
LPCTSTR lpszObjectText = _T("Lorem ipsum dolor sit amet, consectetur adipiscing
elit.");
WEB_STORAGE_OBJECT webObject;

if (WebPutTextObject(hStorage, lpszObjectLabel, lpszObjectText, -1, &webObject))
{
    // The object was created or replaced, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _ftprintf(stderr, _T("Unable to store \"%s\": %s\n"), lpszObjectLabel,
szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCompareText, WebGetObjectEx, WebGetTextObject, WebPutFileEx, WebPutObjectEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebRegisterAppId Function

```
BOOL WINAPI WebRegisterAppId(
   LPCTSTR lpszAppId
);
```

The **WebRegisterAppId** function registers a unique application identifier with the server.

Parameters

lpszAppId

A null-terminated string which identifies the application requesting access. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the function will fail. See the remarks below on the recommended method for identifying your application.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebRegisterAppId** function registers an application ID with the server which uniquely identifies the application that is requesting access to the storage container. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

It is recommended that you use a standard format for the application ID that consists of your company name, application name and optionally a version number. For example:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

It is important to note that with these two example IDs, although they are similar, they reference two different applications. Objects stored using the first ID will not be accessible using the second ID. If you want to store objects that should be shared between all versions of the application, it is recommended that you use the first form, without the version number. If you want to store objects that should only be accessible to a specific version of your application, then it is recommended that you use the second form that includes the version number.

It is safe to call this function with an application ID that was previously registered. If the provided application ID has already been registered, this function will succeed. You can choose to call this function every time your application starts to ensure the application has been registered correctly.

If you no longer wish to use an application ID you have previously registered, you can call the **WebUnregisterAppId** function. Exercise caution when unregistering an application. This will cause all objects stored using that ID to be deleted by the storage server. Once an application ID has been unregistered, the operation is permanent. Calling **WebUnregisterAppId** and then **WebRegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

The application ID is intended to be an application defined human-readable string that uniquely

identifies your application. If you want to obtain the internal storage ID associated with your application, use the **WebGetStorageId** function. The storage ID is a fixed-length string of letters and numbers guaranteed to be unique across all applications that you register.

It is not required for your application to create a unique application ID. Each storage account has a default internal application ID named **SocketTools.Storage.Default**. This default ID is used if a NULL pointer or an empty string is specified to functions like **WebOpenStorage**. It is intended to identify storage available to all applications that you create.

To enumerate the application IDs that you have created, use the **WebGetFirstApplication** and **WebGetNextApplication** functions.

Example

```
 HSTORAGE hStorage = INVALID_HANDLE;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };
BOOL bSuccess = FALSE;

// Register the application ID which identifies the application
if (!WebRegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
hStorage = WebOpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);

if (hStorage == INVALID_HANDLE)
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}

// Upload a local file to the storage server and return information
// about the stored object when it completes
bSuccess = WebPutFileEx(hStorage,
                        lpszLocalFile,
                        lpszObjectLabel,
                        NULL,
                        WEB_OBJECT_DEFAULT,
                        0,
                        &webObject);

if (bSuccess)
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %I64u\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
```

```
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _tprintf(_T("WebPutFileEx failed, error 0x%08lX (%s)\n"), dwError, szError);
}

// Release the handle allocated for this storage session
WebCloseStorage(hStorage);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCloseStorage, WebOpenStorage, WebGetFirstApplication, WebGetNextApplication, WebGetStorageId, WebUnregisterAppId, WebValidateAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebRegisterEvent Function

```
INT WINAPI WebRegisterEvent(
  HSTORAGE hStorage,
  UINT nEventId,
  WEBEVENTPROC lpEventProc,
  DWORD_PTR dwParam
);
```

The **WebRegisterEvent** function registers an event handler for the specified event.

Parameters

hStorage

A handle to the storage container.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. This parameter cannot be zero.

Value	Description
WEB_EVENT_CONNECT	The connection to the storage server has been established and the session has been authenticated. This is the first event that occurs when initiating an operation to create or retrieve a storage object.
WEB_EVENT_DISCONNECT	The connection to the storage server has been closed and the session is terminating. This is the last event that occurs after completing an operation to create or retrieve a storage object.
WEB_EVENT_READ	The contents of an object is being read from the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of retrieving the object contents from the container.
WEB_EVENT_WRITE	The contents of an object is being written to the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of submitting the object contents to the container.
WEB_EVENT_TIMEOUT	The operation has exceeded the specified timeout period. The application may attempt to retry the operation or report an error to the user. This event typically indicates a connectivity problem with the storage server.
WEB_EVENT_CANCEL	The operation has been canceled. This event occurs after the application calls **WebCancelTransfer** while an object is being stored or retrieved.
WEB_EVENT_PROGRESS	A storage operation is in progress. This event

	periodically occurs as the contents of a storage object is being read or written from the container. To retrieve information about the status of the operation, the application should register a handler for this event and call the **WebGetTransferStatus** function from within that handler.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WebEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function specified by ***lpEventProc***. If the application targets x86 (32-bit) platforms, this parameter must be a 32-bit unsigned integer. If the application targets x64 (64-bit) platforms, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebRegisterEvent** function associates a callback function with a specific event. The event handler is a **WebEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the storage handle, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

Events are only generated as the result of a call to the **WebGetFile**, **WebGetFileEx**, **WebPutFile** and **WebPutFileEx** functions. Other storage functions will not generate event notifications.

This function is typically used to register an event handler that is invoked while the contents of a storage object is being transferred. The WEB_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one WEB_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebCancelTransfer, WebEventProc, WebGetTransferStatus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebRenameObject Function

```
BOOL WINAPI WebRenameObject(
   HSTORAGE hStorage,
   LPCTSTR lpszOldLabel,
   LPCTSTR lpszNewLabel
);
```

The **WebRenameObject** function changes the label associated with the storage object.

Parameters

hStorage

A handle to the storage container.

lpszOldLabel

A null-terminated string which specifies the name of the existing storage object to be renamed. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A null-terminated string which specifies a new label for the storage object being moved. This parameter may not be NULL or point to an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebRenameObject** function is used to change the label for an existing storage object. Although storage object labels are similar to Windows file names, they are case-sensitive. When renaming an object, your application must specify the original label name exactly as it was created. The object label cannot contain wildcard characters.

To duplicate an existing storage object, use the **WebCopyObject** function. If you need to move the object to a different storage container, use the **WebMoveObject** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebCopyObject, WebDeleteObject, WebMoveObject

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebResetStorage Function

```
BOOL WINAPI WebResetStorage(
   HSTORAGE hStorage
);
```

The **WebResetStorage** function resets the application storage container and deletes all stored objects.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The storage container contains information for each of the objects that have been stored using the handle returned by the **WebOpenStorage** function. Each of these objects are associated with both the application ID and the storage type that was specified by the caller. This function instructs the server to reset the container back to its initial state, deleting all of the objects that were stored in it.

Exercise caution when using this function. The reset operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebDeleteObject, WebOpenStorage, WebRegisterAppId, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebResetStorageEx Function

```
BOOL WINAPI WebResetStorageEx(
   HSTORAGE hStorage,
   DWORD dwStorageType,
   LPVOID lpvReserved,
   LPDWORD lpdwObjects
);
```

The **WebResetStorageEx** function resets the specified storage container and deletes all stored objects.

Parameters

hStorage

A handle to the storage container.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpvReserved

A reserved parameter that should always be NULL. If your project targets the x64 platform, this

must be a 64-bit pointer with a value of zero. If this parameter is not a NULL pointer, the function will fail.

lpdwObjects

A pointer to an unsigned integer value which will contain the number of objects that were deleted when the contents of the container were expunged.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The storage container contains information for each of the objects that have been stored using the handle returned by the **WebOpenStorage** function. Each of these objects are associated with both the application ID and the storage type that was specified by the caller. This function instructs the server to reset the container back to its initial state, deleting all of the objects that were stored in it.

Exercise caution when using this function. The reset operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

Unlike the **WebResetStorage** function which deletes all of the objects in the container opened using **WebOpenStorage**, this function enables you reset the contents of a different container without explicitly opening it. For example, if you used **WebOpenStorage** to open the WEB_STORAGE_GLOBAL container, you could call this version of the function using that handle, but specifying the WEB_STORAGE_USER container to remove all of the objects created by the current user without affecting the stored objects in the global container.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebDeleteObject, WebOpenStorage, WebRegisterAppId, WebResetStorage, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebSetLastError Function

```
VOID WINAPI WebSetLastError(
   DWORD dwErrorCode
);
```

The **WebSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WEB_ERROR. Those functions which call **WebSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **WebGetLastError** function. The use of **WebGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetErrorString, WebGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebSetStorageTimeout Function

```
INT WINAPI WebSetStorageTimeout(
    HSTORAGE hStorage,
    INT nSeconds
);
```

The **WebSetStorageTimeout** function sets the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hStorage

A handle to the storage container.

nSeconds

The number of seconds to wait for a storage operation to complete.

Return Value

If the function succeeds, the return value is the previous timeout period set for this storage handle. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The default timeout period is 10 seconds. The minimum timeout period is 5 seconds, and the maximum timeout period is 60 seconds. Values outside of this range will be normalized internally and will not cause the function to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebGetStorageTimeout, WebOpenStorage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebUninitialize Function

```
VOID WINAPI WebUninitialize();
```

The **WebUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **WebInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **WebUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **WebUninitialize** for every successful call to **WebInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib

See Also

WebCloseStorage, WebInitialize, WebOpenStorage, WebRegisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebUnregisterAppId Function

```
BOOL WINAPI WebUnregisterAppId(
    LPCTSTR lpszAppId
);
```

The **WebUnregisterAppId** function unregisters the application identifier and deletes all associated storage objects.

Parameters

lpszAppId

A null-terminated string which specifies the application ID to be deleted. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebUnregisterAppId** function deletes the internal storage identifier associated with the application ID and revokes all access tokens that were granted for the application. This operation is immediate and permanent.

Exercise caution when using this function. This will permanently delete all objects that were stored for the specified application. Calling **WebUnregisterAppId** and then **WebRegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

This function cannot be used to unregister the default storage application identifier `SocketTools.Storage.Default`. If this ID is specified, the function will fail with an error indicating that the ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebOpenStorage, WebRegisterAppId, WebResetStorage, WebValidateAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebUploadFile Function

```
BOOL WINAPI WebUploadFile(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPWEB_STORAGE_OBJECT lpObject
);
```

The **WebUploadFile** function uploads the contents of a local file and creates or overwrites a storage object.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server. This parameter cannot be NULL or an empty string. Refer to the **WebValidateLabel** function for more information about object labels.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebUploadFile** function uploads the contents of a local file and either creates a new storage object, or replaces an object if one already exists with the same label. The object will be created in the storage container WEB_STORAGE_GLOBAL using the application ID `SocketTools.Storage.Default`. This is the same default container used by the **WebDownloadFile** function.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

Unlike the **WebPutFile** and **WebPutFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebUploadFile**.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

The **WebUploadFileEx** function provides similar functionality with a more complex interface that supports custom application IDs, additional options and the ability to provide a callback function that is invoked during the upload process.

Example

```
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the global storage container
if (WebUploadFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // The object was uploaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be uploaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFile, WebPutFile, WebUploadFileEx, WebValidateLabel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebUploadFileEx Function

```
BOOL WINAPI WebUploadFileEx(
   LPCTSTR lpszLocalFile,
   LPCTSTR lpszObjectLabel,
   LPCTSTR lpszAppId,
   DWORD dwStorageType,
   DWORD dwAttributes,
   DWORD dwReserved,
   DWORD dwTimeout,
   LPWEB_STORAGE_OBJECT lpObject,
   WEBEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **WebUploadFileEx** function uploads the contents of a local file and creates or overwrites a storage object.

Parameters

lpszLocalFile

A null-terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A null-terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A null-terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.

WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bit flags using a bitwise OR operation:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **WebEnumObjects** function. The object can only be accessed directly when specifying its label.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

dwTimeout

An unsigned integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a WEB_STORAGE_OBJECT structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WebEventProc** callback function. If this parameter is NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback function specified by ***lpEventProc***. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If

the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebUploadFileEx** function uploads the contents of a local file and either creates a new storage object, or replaces an object if one already exists with the same label. The **WebUploadFile** function provides a simpler interface that defaults to uploading an object to the global storage container.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

Unlike the **WebPutFile** and **WebPutFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebUploadFileEx**.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are uploading a large file and want your application to receive progress updates during the data transfer, provide a pointer to a callback function as the *lpEventProc* parameter. That function will receive event notifications as the data is being uploaded.

Example

```
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the global storage container
if (WebUploadFileEx(lpszLocalFile,
                    lpszObjectLabel,
                    lpszAppId,
                    WEB_STORAGE_GLOBAL,
                    WEB_OBJECT_DEFAULT,
                    0,
                    &webObject,
                    NULL, 0))
{
    // The object was uploaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be uploaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebDownloadFile, WebGetFileEx, WebPutFileEx, WebUploadFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebValidateAppId Function

```
BOOL WINAPI WebValidateAppId(
   LPCTSTR lpszAppId
);
```

The **WebValidateAppId** function validates the specified application identifier.

Parameters

lpszAppId

A null-terminated string which specifies the application ID to be validated. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebValidateAppId** function is used to determine if the specified application identifier is valid and has been previously registered using the **WebRegisterAppId** function. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebRegisterAppId, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebValidateLabel Function

```
BOOL WINAPI WebValidateLabel(
   LPCTSTR lpszObjectLabel
);
```

The **WebValidateLabel** function validates the specified object label.

Parameters

lpszObjectLabel

A null-terminated string which specifies the object label to be validated. This parameter cannot be NULL or point to a zero-length string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

Object labels are similar to Windows file names, except they are case-sensitive. The maximum length of a label string is 512 characters, including the terminating null character. Leading and trailing whitespace (spaces, tabs, linebreaks, etc.) are ignored in label names.

Illegal characters include ASCII and Unicode control characters 1 through 31, single quotes (39), double quotes (34), less than symbol (60), greater than symbol (62), pipe (124), asterisk (42) and question mark (63). A null character (0) specifies the end of the label and any subsequent characters are ignored. It is not possible to embed null characters in the label name.

Label names may contain forward slash (47) characters and backslash (92) characters, however it is important to note that objects are not stored in a hierarchical structure. An application can create its own folder-like structure to the labels it creates, but this structure is not imposed or enforced by the library.

If the application is built to use Unicode, labels can contain Unicode characters which are internally encoded as UTF-8. This is important to consider if you have an project built using a multi-byte (ANSI) character set and it needs to access an object that was created using Unicode characters. In that case, the ANSI application must be prepared to handle UTF-8 encoded names and display them appropriately.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswebv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebRegisterAppId, WebUnregisterAppId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Web Services Data Structures

- SYSTEMTIME
- WEB_LOCATION
- WEB_STORAGE_APPLICATION
- WEB_STORAGE_QUOTA
- WEB_STORAGE_OBJECT
- WEB_STORAGE_TRANSFER

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: windows.h.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_LOCATION

The **WEB_LOCATION** structure contains information about the current location of the local computer system.

```
typedef struct _WEB_LOCATION
{
    TCHAR   szLocationId[40];
    TCHAR   szIPAddress[46];
    UINT    nAutonomousSytemNumber;
    TCHAR   szOrganization[64];
    TCHAR   szRegionName[64];
    UINT    nRegionCode;
    TCHAR   szCountryName[64];
    TCHAR   szCountryAlpha2[4];
    TCHAR   szCountryAlpha3[4];
    UINT    nCountryCode;
    TCHAR   szSubdivision[64];
    TCHAR   szSubdivisionCode[4];
    TCHAR   szCityName[64];
    TCHAR   szPostalCode[32];
    TCHAR   szCoordinates[32];
    TCHAR   szTimezone[64];
    TCHAR   szTzShortName[16];
    LONG    nTimezoneOffset;
    DOUBLE  dLatitude;
    DOUBLE  dLongitude;
    SYSTEMTIME stLocalTime;
} WEB_LOCATION, *LPWEB_LOCATION;
```

Members

szLocationId

A null-terminated string which contains contains a string of hexadecimal characters which uniquely identifies the location for this computer system. This value is used internally by the location service, and may also be used by the application for its own purposes. If this value changes in subsequent queries, it indicates the external IP address for the local system has changed.

lpszIPAddress

A null-terminated string which contains the external IP address for the local system. If the system has been assigned multiple IP addresses, it reflects the address of the interface that was used to establish a connection with the SocketTools server. If the connection is made through a Virtual Private Network (VPN) it will use that assigned IP address. If a connection is made through a proxy server, the IP address may the address of the proxy rather than the local host, depending on how the connection is made.

nAutonomousSystemNumber

An unsigned integer value which is used to uniquely identify a global network (autonomous system) that is connected to the Internet. This number is assigned by regional registries and used by large networks, such as Internet Service Providers, for exchanging routing information with one another. This value can be used to determine the ownership of a particular network.

szOrganization

A null-terminated string which identifies the organization associated with the local system's external IP address. For residential end-users this is typically the name of their Internet Service provider, however it may also identify a private company such as Microsoft, Google or Amazon. Because of

the nature of how this information is updated, the organization names can change over time due to acquisitions or changes of ownership. If the owner of the network cannot be determined, this member may contain an empty string.

szRegionName

A null-terminated string which identifies the region in which the external IP address is located. This refers to a broad geographical area, such as "North America" or "Southeast Asia" and uses the conventions for supranational regions as defined by UN M49 codes. These names will always be in English, regardless of the current system locale.

nRegionCode

An unsigned integer value which identifies the geographical region in which the external IP address is located. This value corresponds to the name returned in the ***szRegionName*** structure member. The numeric values use the UN M49 standard established by the United Nations Statistics Division.

szCountryName

A null-terminated string which contains the full name of the country in which the external IP address is located. These names will always be in English, regardless of the current system locale.

szCountryAlpha2

A null-terminated string which contains the ISO 3166-1 alpha-2 code for the country the external IP address is located in. This is a two-letter country code established by the International Organization for Standardization (ISO). For example, the code for the United States is "US".

szCountryAlpha3

A null-terminated string which contains the ISO 3166-1 alpha-3 code for the country the external IP address is located in. This is a three-letter country code established by the International Organization for Standardization (ISO). For example, the code for the United States is "USA".

nCountryCode

An unsigned integer value which identifies the country where the external IP address is located. These codes can be up to three digits (usually displayed with leading zeros as necessary) and correspond to the country codes assigned by the United Nations. For example, the code for the United States is 840. It is important to note that these are not international dialing codes and should not be used with telephony applications.

szSubdivision

A null-terminated string which identifies the geopolitical subdivision within a country where the external IP address is located. In the United States, this will contain the full name of the state or commonwealth. In Canada, this will contain the name of the province or territory. These names will always be in English, regardless of the current system locale.

szSubdivisionCode

A null-terminated string which is either a two- or three-letter code which identifies a geopolitical subdivision within the country where the external IP address is located. These codes are defined by the ISO 3166-2 standard. For example, the code for the state of California in the United States is "CA". For specificity, these subdivision codes are often combined with the ISO 3166-1 alpha-2 or alpha-3 code for the country the subdivision is located within. Using the previous example of California, ISO alpha-2 code and subdivision code would be combined as "US-CA".

szCityName

A null-terminated string which identifies the city in which the external IP address is located. These names will always be in English, regardless of the current system locale. If the city name cannot be determined, this member may contain an empty string.

szPostalCode

A null-terminated string which contains the postal code associated with the area where the IP address is located. In the United States, this is a 5-digit numeric code. In Canada, this will contain the forward sortation area (the first three characters of the six character postal code). Local delivery portions of a postal code (such as the ZIP+4 code in the United States, or the local delivery unit in Canada) are not included. This information will be most accurate within the geographic region of North America. Postal codes are locale specific, and this structure member may contain an empty string if the postal code for the location cannot be determined.

szCooordinates

A null-terminated string which specifies the location expressed using the Universal Transverse Mercator (UTM) coordinate system with the WGS-84 ellipsoid. UTM coordinates are commonly used with the Global Positioning System (GPS) and are comprised of three parts: the zone, the easting (the eastward-measured distance or ***x***-coordinate) and the northing (the northward-measured distance or ***y***-coordinate). An example of a string value returned in this structure member would be "14S 702089E 3646476N".

szTimezone

A null-terminated string which specifies the full time zone name in which the external IP address is located. These names are defined by the Internet Assigned Numbers Authority (IANA) and have values like "America/Los_Angeles" and "Europe/London". These time zones may also be defined as "Etc/GMT+10" if there is not a regional name associated with the time zone.

szTzShortName

A null-terminated string which specifies the abbreviated time zone code in which the external IP address is located. If daylight savings time is used within the time zone, then this value can change based on whether or not daylight savings is in effect. For example, if the IP address is located within the Pacific time zone in the United States, this will return "PDT" when daylight savings is in effect and "PST" when it is not. If a short time zone code cannot be determined, a value such as "UTC+9" may be returned, indicating the number of hours ahead or behind UTC.

nTimezoneOffset

A signed integer which specifies the number of seconds east or west of the prime meridian (UTC). A positive value indicates a time zone that is east of the prime meridian and a negative value indicates a time zone that is west of the prime meridian. For example, the Pacific time zone in the western United States during daylight savings time would have a value of -25200 (7 hours).

dLatitude

A floating point value which specifies the latitude of the location in decimal format. A positive value indicates a location that is north of the equator, while a negative value is a location that is south of the equator.

dLongitude

A floating point value which specifies the longitude of the location in decimal format. A positive value indicates a location that is east of the prime meridian, while a negative value is a location that is west of the prime meridian.

stLocalTime

A SYSTEMTIME structure that contains information about the current date and time at the location, adjusted for its time zone and whether or not it's in daylight savings time.

Remarks

Use the **WebGetLocation** function to populate this structure with information about the physical

location of the local computer system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

WebGetLocation

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_APPLICATION

The **WEB_STORAGE_APPLICATION** structure contains information about a registered application.

```
typedef struct _WEB_STORAGE_APPLICATION
{
    TCHAR      szAppId[64];
    TCHAR      szApiKey[64];
    TCHAR      szLuid[64];
    DWORD      dwTokens;
    DWORD      dwReserved;
    SYSTEMTIME stCreated;
    SYSTEMTIME stUpdated;
} WEB_STORAGE_APPLICATION, *LPWEB_STORAGE_APPLICATION;
```

Members

szAppId

A null-terminated string which contains the application ID that was registered using the **WebRegisterAppId** function.

szApiKey

A null-terminated string which contains the API key value that is used internally by the storage services. The AppID is effectively the human readable alias for this key value used to identify the stored objects for the application. This value is guaranteed to be unique across all applications registered with the storage service.

szLuid

A null-terminated string which specifies a locally unique value associated with the registered application. This value is used internally by the storage service and guaranteed to be unique to the storage account that has registered the application. Note that there are no functions that currently accept the application LUID as parameter, but you may choose to use this value in your own application for other purposes.

dwTokens

An unsigned integer value which specifies the number of access tokens associated with the registered application. Typically there is only one access token associated with a given AppID at any one time, although it is possible that the API may allocate additional access tokens under some circumstances.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

stCreated

A SYSTEMTIME structure that specifies the date and time the AppID was registered. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stUpdated

A SYSTEMTIME structure that specifies the date and time the AppID object was last updated. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

This structure is used in conjunction with the **WebEnumApplications** function which is used to enumerate all of the registered applications for the storage account. It is also used with the

WebGetFirstApplication and **WebGetNextApplication** functions which provide an alternative method for enumerating registered applications.

To adjust the creation and update times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** function. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_OBJECT

The **WEB_STORAGE_OBJECT** structure contains information about an individual storage object.

```
typedef struct _WEB_STORAGE_OBJECT
{
    TCHAR      szObjectId[64];
    TCHAR      szLabel[512];
    TCHAR      szDigest[128];
    TCHAR      szContent[128];
    DWORD      dwAttributes;
    DWORD      dwObjectSize;
    SYSTEMTIME stCreated;
    SYSTEMTIME stModified;
} WEB_STORAGE_OBJECT, *LPWEB_STORAGE_OBJECT;
```

Members

szObjectId

A null-terminated string which contains the unique identifier associated with this object. Object IDs are guaranteed to be unique for each storage object that is created by the application. The maximum length of an object ID is 64 characters, including the terminating null character.

szLabel

A null-terminated string which contains the label assigned to the object by the application. Object labels are case-sensitive and must be unique for each object. An application uses labels to reference an object with a human-recognizable name, rather than referencing them by their object ID. The maximum length of an object label is 512 characters, including the terminating null character.

szDigest

A null-terminated string which specifies the digest of the object contents, computed using an SHA-256 hash. The maximum length of the ***szDigest*** string is 128 characters, including the terminating null character. However, the digest value is always represented as a string of hexadecimal numbers that is exactly 64 characters long. It is important to note that even a zero-length object will have a digest, which is the standard SHA-256 NULL hash value.

szContent

A null-terminated string which specifies the MIME content type for the storage object. The content type is determined by the object label and evaluating the contents of the object. It is also possible for the application to explicitly specify the content type of the object when it is created.

dwAttributes

An unsigned integer value that specifies the attributes for the storage object. The object attributes are comprised of one or more bit flags:

Value	Description
WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
WEB_OBJECT_READONLY	A read-only object that can only be read by the application.

	Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the **WebEnumObjects** function. The object can only be accessed directly when specifying its label.

dwObjectSize

An unsigned integer value that specifies the size of the storage object in bytes. The maximum size of an individual object is determined by the storage quota limits established for the account.

stCreated

A SYSTEMTIME structure that specifies the date and time the storage object was created. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stModified

A SYSTEMTIME structure that specifies the date and time the storage object was last modified. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

The object content type will always be in the format **type/subtype** where the type specifies a common media type (e.g.: text, audio, video, etc.) and subtype specifies the specific content. The most common content type for text files is **text/plain**. If the content type is unknown, the default content type is **application/octet-stream**.

Text objects may also optionally include the character encoding as part of the content type. For example, if an object contains UTF-8 encoded text, the content type may be returned as **text/plain; charset=utf-8**. If your application is parsing the content types, you must check if a character encoding was also included in the value. Text objects that do not specify an encoding either contain ASCII or text which uses the system code page. Unicode text will always be stored using UTF-8 encoding.

To adjust the object creation and modification times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** function. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_QUOTA

The **WEB_STORAGE_QUOTA** structure contains information about the current storage account data usage and limits.

```
typedef struct _WEB_STORAGE_QUOTA
{
    DWORD     dwObjects;
    DWORD     dwObjectLimit;
    DWORD     dwObjectSize;
    ULONGLONG ulBytesUsed;
    ULONGLONG ulBytesFree;
    ULONGLONG ulStorageLimit;
} WEB_STORAGE_QUOTA, *LPWEB_STORAGE_QUOTA;
```

Members

dwObjects

An unsigned integer value which specifies the number of storage objects allocated for the account. This value may not exceed the total number of objects specified by the ***dwObjectLimit*** member.

dwObjectLimit

An unsigned integer value which specifies the maximum number of storage objects that may be created. In addition to the limit on the total amount of storage that may be used, there is a limit on the total number of objects that may be created by all applications.

dwObjectSize

An unsigned integer value which specifies the maximum size of an individual storage object. In addition to a limit on the total amount of storage used and the number of objects created, each object stored by the application cannot exceed this size.

ulBytesUsed

An unsigned 64-bit integer value which specifies the total number of bytes of data allocated for all storage objects. This value may not exceed the total number of bytes of storage available, which is specified by the ***ulStorageLimit*** member.

ulBytesFree

An unsigned 64-bit integer value which specifies the number of bytes available for the storage of new objects. This value reflects the total amount of available storage across all applications registered with the development account. If this value is zero, your storage account has reached its storage limit.

ulStorageLimit

An unsigned 64-bit integer value which specifies the maximum number of bytes of data storage available. This limit applies to all applications registered with the development account.

Remarks

Storage quota limits are assigned for each SocketTools development account. The **WebGetStorageQuota** function will populate this structure with information about the limits on your account. Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

These values do not represent limits on storage usage by a specific application. Quotas limits apply to all applications that are registered with the development account, which is identified with the runtime

license key passed to the **WebInitialize** function.

If your storage quota has been exceeded, either because the total number of objects or the total bytes of storage has reached their limit, your applications will be unable to create new objects. Your application can continue to access existing objects, regardless of your current quota limits.

To free storage space, use the **WebDeleteObject** function to delete individual storage objects that are no longer needed by your application, or use the **WebResetStorage** function to delete all objects in a container.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

See Also

WebGetStorageQuota

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEB_STORAGE_TRANSFER

This structure is used by the **WebGetTransferStatus** function to return information about a data transfer in progress.

```
typedef struct _WEB_STORAGE_TRANSFER
{
   TCHAR     szObjectLabel[512];
   DWORD     dwStorageType;
   DWORD     dwBytesTotal;
   DWORD     dwBytesCopied;
   DWORD     dwBytesPerSecond;
   DWORD     dwTimeElapsed;
   DWORD     dwTimeEstimated;
} WEB_STORAGE_TRANSFER, *LPWEB_STORAGE_TRANSFER
```

Members

szObjectLabel

A null-terminated string that specifies the label for the object that is being retrieved, created or replaced.

dwStorageType

An integer value that identifies the storage container type. This will be one of the following values:

Value	Description
WEB_STORAGE_GLOBAL	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwBytesTotal

An unsigned integer which specifies the total number of bytes that will be transferred. If the object is being downloaded to the local host, this is the size of the stored object. If the data is being uploaded from the local host to be stored on the server, it is the size of the buffer or local file.

dwBytesCopied

An unsigned integer which specifies the total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebView Browser Library

The SocketTools WebView library provides an embedded Chromium-based web browser that can be integrated into desktop applications. The library is implemented using the Microsoft Edge WebView2 runtime and provides a much simpler API which can be used with most languages, without requiring direct use of the COM-based interfaces in the WebView2 SDK.

Reference

- Functions
- Constants
- Data Structures
- Error Codes

Library Information

File Name	CSHTMV12.DLL
Version	12.0.3024.1286
LibID	175ACE49-FE5D-46C8-A0EF-5188C97B993D
Import Library	CSHTMV12.LIB
Dependencies	None

Overview

The SocketTools WebView library makes it possible to embed modern web content into Windows applications using a simple procedural interface. Developers can work entirely with standard data types and function calls, avoiding the complexity of COM programming or .NET interop layers. The API is designed to be language-agnostic and can be called from C or C++, as well as other environments that support invoking functions from a dynamic link library such as Object Pascal and Visual Basic. Applications can use a traditional Windows GUI or be console-based, and functions are thread-safe where appropriate.

Because the library encapsulates the creation of the WebView2 environment and its associated controller, applications require only minimal setup. Developers can host a browser window, execute scripts, or capture output with just a few lines of code. Advanced features such as asynchronous execution, response header enumeration, and event callbacks are supported, but optional, making it easy to start simple and grow into more complex scenarios.

Applications are not limited to navigating to external websites. You can dynamically generate HTML content within your program and load it directly into the browser, making it possible to render reports, status pages, or other application data without creating temporary files. The browser also supports displaying local content using `file://` URLs, including HTML, images, PDF documents, and other formats that WebView2 can natively render. This makes it easy to blend remote and local resources within the same user interface.

When connecting to web servers, the control uses the underlying WebView2 runtime and supports modern versions of HTTP, including HTTP/1.1, HTTP/2, and HTTP/3 where available. Connections are secured with Transport Layer Security (TLS), with support for TLS 1.2 and TLS 1.3 by default, and certificate validation is performed automatically by the runtime. This ensures that applications embedding the browser benefit from the same security standards as Microsoft Edge, including

enforcement of current best practices for encryption and certificate handling.

Unlike most SocketTools libraries which return pseudo-handles to reference client sessions, the WebView library uses window handles (HWND) for nearly all of its functions. The only exceptions are the functions used to create or attach to a WebView instance. This design provides a direct connection between the library and the underlying windowing system, and ensures consistent behavior with standard Windows APIs. Even in console applications, the library creates a host window which provides the handle used with the other functions. Using window handles also makes integration with GUI frameworks such as MFC, WinForms, or WPF more natural, since those frameworks are already designed around standard Windows window management.

Requirements

It is not necessary to download the Microsoft.Web.WebView2 SDK NuGet package and include it in your project to use this library. However, the end-user system must have the current version of the Edge WebView2 Runtime installed. On Windows 11 and Windows 10 version 2004 (build 19041) and later, the runtime is considered part of the operating system and is updated automatically. On earlier platforms such as Windows 7, Windows 8, and Windows 10 builds prior to 19041, you may need to redistribute or install the WebView2 runtime separately. The library includes functions which can tell you if the runtime is installed and what version is available.

For Windows 7 and Windows 8, the last version of the WebView2 runtime available is 109.0.1518.140. This version is supported by SocketTools, however it is important to note that Microsoft is no longer updating the control on those platforms. This means it will no longer receive security updates or other improvements; some features may work differently on older platforms, and newer features will not be available.

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

In addition to the SocketTools library, the WebView2 Runtime must be present on the target system. On current versions of Windows 10 and Windows 11 the runtime is treated as part of the operating system and is updated automatically. On earlier platforms, you may need to redistribute or install the runtime using Microsoft's evergreen installer. We recommend always using the evergreen runtime rather than the fixed version, as it ensures security updates and feature improvements are delivered through Windows Update without requiring changes to your application.

The WebView2 Runtime is available as a separate download from Microsoft. For current links and redistribution guidelines, please refer to the SocketTools knowledge base or Microsoft's official documentation.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebView Functions

Function	Description
WebViewAsyncExecute	Execute JavaScript asynchronously in the current document.
WebViewAttachWindow	Attach the browser to an existing host window.
WebViewAttachWindowEx	Attach the browser to an existing host window with additional options.
WebViewCancelDownload	Cancel the specified file download.
WebViewCancelNavigation	Cancel the current navigation and stop loading.
WebViewClearCache	Clear the cache of viewed pages for the current user profile.
WebViewClearCookies	Delete cookies for the current user profile or session.
WebViewClearHeaders	Clear the request and/or response headers for the current session.
WebViewClearHistory	Clear the navigation history for the current session.
WebViewCreateWindow	Create an instance of the browser in a new window.
WebViewCreateWindowEx	Create an instance of the browser in a new window with additional options.
WebViewDestroyWindow	Destroy a host window and release resources allocated for the browser session.
WebViewDetachWindow	Detach the browser from its host window without destroying the host window.
WebViewDisableEvents	Disable event handling for the specified browser.
WebViewEnableEvents	Enable event handling for the specified browser.
WebViewEnumDownloads	Enumerate the file identifiers for all downloads during the current session.
WebViewEventProc	An application-defined callback function that processes events generated by the browser
WebViewExecute	Execute JavaScript and return the result synchronously.
WebViewFindWindow	Locate the associated host or child window handle for an instance.
WebViewFreezeEvents	Suspend or re-enable event notifications for the specified browser.
WebViewGetAllOptions	Return the current option flags for the browser instance.
WebViewGetBackgroundColor	Get the background color used behind the page content.
WebViewGetBrowserVersion	Get the installed browser runtime version information.
WebViewGetChildWindow	Get the handle of the internal browser child window.
WebViewGetContentType	Return the Content-Type from the last response, if available.
WebViewGetCurrentUrl	Get the URL of the current document.
WebViewGetDocumentText	Return the content for the current document.
WebViewGetDocumentTitle	Return the document title.
WebViewGetDownloadInfo	Return information about the status of the specified download.
WebViewGetDownloadPath	Return the full path for the downloaded file.
WebViewGetErrorString	Return a description for the specified error code.
WebViewGetFilteredText	Return the current document text without HTML formatting.
WebViewGetFirstHeader	Begin enumerating response headers and return the first name/value pair.
WebViewGetFocus	Return whether the browser control currently has input focus.
WebViewGetHeader	Get the value of a specific response header by name.

WebViewGetHeaderCount	Return the number of response headers from the last navigation.
WebViewGetHttpStatus	Return the HTTP status code from the last completed navigation.
WebViewGetInterface	Returns a raw COM interface pointer to the WebView2 browser control.
WebViewGetLastError	Return the last error code.
WebViewGetMessage	Retrieve the next pending message sent from the current document.
WebViewGetNavigationStatus	Return the current navigation state.
WebViewGetNavigationTime	Return the elapsed time for the last navigation.
WebViewGetNextHeader	Continue header enumeration and return the next name/value pair.
WebViewGetOption	Checks whether a specific option flag is enabled.
WebViewGetResult	Return the result from the most recent completed script execution.
WebViewGetSecurityInfo	Return certificate and host security details for the last navigation.
WebViewGetSecurityLevel	Return the evaluated security level for the last navigation.
WebViewGetTimeout	Get the number of seconds to wait for a navigation to complete.
WebViewGetUserAgent	Return the current User-Agent value for this instance.
WebViewGetUserDataFolder	Return the full path to the current user data folder.
WebViewGetZoomLevel	Return the current zoom level as a percentage.
WebViewGoBack	Navigate backwards through the browser history.
WebViewGoForward	Navigate forwards through the browser history.
WebViewInEvent	Determine if the current thread is executing code within an event handler.
WebViewInitialize	Initialize the library and validate a license key at runtime.
WebViewIsVisible	Determine if the browser window is currently visible.
WebViewKeyEventHandled	Set an accelerator key event as handled to suppress default processing.
WebViewNavigateUrl	Navigate to the specified URL or local file path.
WebViewNormalizeUrl	Normalize a URL or path for safe, consistent navigation.
WebViewPostMessage	Post a string or JSON message to the current page.
WebViewPrintDocument	Display the browser print dialog for the current document.
WebViewRedrawWindow	Force the host window and browser control to redraw themselves.
WebViewReloadDocument	Reload the current document and optionally clear the cache.
WebViewResetBrowser	Reset the embedded runtime control.
WebViewRuntimeAvailable	Return whether the required runtime is installed and available.
WebViewSaveDocument	Save the current document to a local file.
WebViewSetBackgroundColor	Set the background color used behind the page content.
WebViewSetBounds	Set the position and size of the browser child window.
WebViewSetDocumentText	Replace the current document with the specified HTML or plain text.
WebViewSetDownloadPath	Change the local file path for the specified download.
WebViewSetFocus	Give keyboard focus to the browser control.
WebViewSetHeader	Set or override a request header for subsequent navigations.
WebViewSetIcon	Set custom large and small icons for the host window.

WebViewSetLastError	Set the last error code.
WebViewSetOption	Enable or disable a specific option.
WebViewSetTimeout	Set the number of seconds to wait for a navigation to complete.
WebViewSetUserAgent	Set the default User-Agent string for this instance.
WebViewSetZoomLevel	Set the zoom level for the current view.
WebViewShowWindow	Show or hide the browser window.
WebViewUninitialize	Terminate use of the library and release global resources.
WebViewWaitForEvent	Wait until a specific event occurs or the timeout period expires.
WebViewWaitUntilClosed	Wait until the host window is closed or destroyed.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewAsyncExecute Function

```
BOOL WINAPI WebViewAsyncExecute(
   HWND hWndHost,
   LPCTSTR lpszJavaScript,
   DWORD_PTR dwParam
);
```

The **WebViewAsyncExecute** function executes a script on the current page, returning the results asynchronously.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszJavaScript

A null-terminated string which specifies the JavaScript that should be executed in the context of the current page being viewed. This parameter is required and cannot be NULL.

dwParam

An optional parameter which can be used to provide context to the script being executed asynchronously. This user-defined value is passed back to the event handler which is invoked when the result from the script is returned back to the application. If this information is not required, this parameter can be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewAsyncExecute** function evaluates a JavaScript expression within the context of the currently loaded page and returns the result asynchronously. Unlike **WebViewExecute**, this function does not block while waiting for the script to complete. Instead, the result is delivered through the `WEBVIEW_EVENT_SCRIPT_COMPLETED` event. The application can use the dwParam value to correlate the completion event with the original request.

The script must be valid JavaScript and should not assume synchronous execution. Results are returned by calling **WebViewGetResult** which will provide you with the response in JSON-encoded form; for example, string values will be quoted. Because script execution is dispatched to the underlying WebView2 control, this function will fail if navigation is still in progress. Applications should always wait for the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event before invoking script execution.

Example

```
// A simple event handler implementation
VOID CALLBACK WebViewEventProc(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    if (uEvent == WEBVIEW_EVENT_SCRIPT_COMPLETED)
    {
        BOOL bSuccess = (BOOL)wParam;

        if (bSuccess)
        {
```

```
            TCHAR szResult[512];
            if (WebViewGetResult(hWnd, szResult, _countof(szResult)))
            {
                // Get the response from the script
                // szResult will be a null-terminated string which may contain
                // a JSON object or JSON string (quoted and escaped if needed)
            }
        }
        else
        {
            // The script could not be executed
        }
    }
}

void RunExample(HWND hWebView)
{
    // Execute a simple script asynchronously
    if (!WebViewAsyncExecute(hWebView, _T("document.title"), 0))
    {
        DWORD dwError = WebViewGetLastError();
        // Handle the error condition
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewExecute, WebViewGetResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewAttachWindow Function

```
BOOL WINAPI WebViewAttachWindow(
   HWND hWndHost,
   LPCTSTR lpszUrl,
   WEBVIEWEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **WebViewAttachWindow** function attaches a new instance of the browser control to a host window.

Parameters

hWndHost

A handle to the window that will serve as a container for the browser control. This may either be a top-level application window or a child window. If the handle does not specify a valid window, the function will fail. This parameter is required and cannot be NULL.

lpszUrl

A null-terminated string which specifies the initial URL the browser should navigate to after the control has been attached to the window. If this parameter is NULL or an empty string, the browser instance will be created with a blank page. The only schemes permitted are `https://`, `http://` and `file://`. If a domain name is specified without a scheme, it will default to using `https://`.

lpEventProc

A callback function which will be invoked whenever browser events occur. If the application does not need to process events, this parameter can be NULL. The callback function will be invoked in the current thread and the executes in the same context as the browser control. For more information, refer to the WebViewEventProc and WebView Events documentation.

dwParam

An optional parameter which can be used to provide additional context to the event handler. This user-defined value is passed back to the callback function when an event occurs. If this information is not required, this parameter can be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewAttachWindow** function binds a new WebView instance to an existing top-level or child window you have already created. The host window must be created on the current UI thread and remain valid for the lifetime of the WebView. Do not call this function from a worker thread or for a window that was created by another thread; if a background thread needs a browser, create a dedicated window on that thread or use **WebViewCreateWindow** instead. Only one WebView instance can be attached to a given host window at a time. If you need to recreate the browser, call **WebViewDetachWindow** before attaching again.

The calling thread must be in a single-threaded apartment (STA). During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), the attach will fail. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

By default, the instance is created with `WEBVIEW_OPTION_ENABLE_DEFAULT`, enabling common browser features (accelerator keys, context menu, basic status UI, zoom, audio, script execution, and script dialogs). If you need a more restrictive experience (such as no context menu enabled), you can use **WebViewAttachWindowEx** to specify an explicit option mask up front.

Your `WEBVIEWEVENTPROC` event callback function is invoked on the same thread that owns the WebView. Because events are raised on the UI thread, long-running work in an event handler can block the user interface and stall navigation. Where possible, defer heavy operations to background threads. You should avoid displaying modal dialogs or message boxes from within event handlers. Also note that some events may fire multiple times during a single navigation. For example, the security state can transition through several levels (unknown, neutral, secure, or warning) before settling on a final value. Applications should be written to handle these changes incrementally.

To present correctly inside a resizable child window, resize the browser view when the host window size changes. The library tracks the host window position but relies on your WM_SIZE handling to drive layout.

Example

```
#include "cstools12.h"

// Windows message handler
static LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// WebView event handler for navigation and security events
static VOID CALLBACK WebViewEvents(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    switch (uEvent)
    {
    case WEBVIEW_EVENT_NAVIGATION_STARTING:
        OutputDebugString(_T("[WebView] Navigation starting\n"));
        break;

    case WEBVIEW_EVENT_NAVIGATION_COMPLETED:
        if (wParam)
            OutputDebugString(_T("[WebView] Navigation completed\n"));
        else
            OutputDebugString(_T("[WebView] Navigation failed\n"));
        break;

    case WEBVIEW_EVENT_SECURITY_CHANGED:
        OutputDebugString(_T("[WebView] Security changed\n"));
        break;
    }
}

INT WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR
lpCmdLine, INT nCmdShow)
{
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        MessageBox(NULL, _T("Unable to initialize the WebView API"), _T("Error"),
MB_ICONEXCLAMATION);
        return 0;
    }

    LPCTSTR kClass = _T("MyWebViewWndClass");
```

```
    WNDCLASS wc{};
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszClassName = kClass;
    RegisterClass(&wc);

    HWND hWnd = CreateWindowEx(0, kClass, _T("WebView"),
                               WS_OVERLAPPEDWINDOW,
                               CW_USEDEFAULT, CW_USEDEFAULT, 1024, 720,
                               NULL, NULL, hInstance, NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // A standard Windows message loop
    MSG uMsg;

    while (GetMessage(&uMsg, nullptr, 0, 0))
    {
        TranslateMessage(&uMsg);
        DispatchMessage(&uMsg);
    }

    WebViewUninitialize();
    return (int)uMsg.wParam;
}

static LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
        // Create a WebView2 browser instance and attach to this window
        if (!WebViewAttachWindow(hWnd, _T("https://example.com"), WebViewEvents,
0))
        {
            MessageBox(NULL, _T("Unable to attach the WebView2 control"),
_T("Error"), MB_ICONEXCLAMATION);
        }
        return 0;

    case WM_SIZE:
        // Make sure the browser window resizes with the window
        {
            RECT rc;
            GetClientRect(hWnd, &rc);
            WebViewSetBounds(hWnd, rc.left, rc.top, rc.right - rc.left, rc.bottom -
rc.top);
        }
        return 0;

    case WM_DESTROY:
        // Detach the WebView2 browser instance from the window
        WebViewDetachWindow(hWnd);
        PostQuitMessage(0);
        return 0;
```

```
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindowEx, WebViewCreateWindow, WebViewCreateWindowEx, WebViewDetachWindow, WebViewEventProc, WebViewInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewAttachWindowEx Function

```
BOOL WINAPI WebViewAttachWindowEx(
   HWND hWndHost,
   DWORD dwOptions,
   LPCTSTR lpszUrl,
   LPCTSTR lpszLanguage,
   LPCTSTR lpszDataFolder,
   WEBVIEWEVENTPROC lpEventProc,
   DWORD dwParam
);
```

The **WebViewAttachWindowEx** function attaches a new instance of the browser control to a host window. This extended version of the function allows the developer to provide additional options, an alternate display language and specify an optional folder where the browser will store session data.

Parameters

hWndHost

A handle to the window that will serve as a container for the browser control. This may either be a top-level application window or a child window. If the handle does not specify a valid window, the function will fail. This parameter is required and cannot be NULL.

dwOptions

One or more option bitflags. For more information, refer to the WebView Options documentation. Most applications should not use a value of zero for this parameter. If no options are specified, the browser instance will be created with a very restrictive set of permissions that can prevent many websites from functioning correctly.

lpszUrl

A null-terminated string which specifies the initial URL the browser should navigate to after the control has been attached to the window. If this parameter is NULL or an empty string, the browser instance will be created with a blank page. The only schemes permitted are `https://`, `http://` and `file://`. If a domain name is specified without a scheme, it will default to using `https://`.

lpszLanguage

A null-terminated string which specifies an alternate language to use when the browser displays a context menu or dialogs. This value can be a standard language tag in BCP-47 format (such as "en-US") or it can be a string which specifies a hexadecimal Windows locale identifier (such as "0x0409"). It is recommended you use the language tag whenever possible, as Microsoft is deprecating the use of locale identifiers (LCIDs) and their associated APIs. If this parameter is a NULL pointer or an empty string, the browser will use the current default language for the local user.

lpszDataFolder

A folder name where the user profile data will be stored. This is the location where the browser will cache viewed pages, store cookies and form data, and where scripts may store session data. This value may be just a name (such as the name of the application) or an absolute path. If the folder name contains environment variables surrounded in percent (%) symbols, they will be replaced with their value in the current user's environment. If folder name does specify an absolute path, the profile will be stored in the current user's LocalAppData folder. If this parameter is NULL or an empty string, a temporary profile will be created for the session and then automatically deleted when the session ends.

lpEventProc

A callback function which will be invoked whenever browser events occur. If the application does not need to process events, this parameter can be NULL. The callback function will be invoked in the current thread and the executes in the same context as the browser control. For more information, refer to the WebViewEventProc and WebView Events documentation.

dwParam

An optional parameter which can be used to provide additional context to the event handler. This user-defined value is passed back to the callback function when an event occurs. If this information is not required, this parameter can be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewAttachWindowEx** function binds a new WebView instance to an existing top-level or child window you have already created. The host window must be created on the current UI thread and remain valid for the lifetime of the WebView. Do not call this function from a worker thread or for a window that was created by another thread; if a background thread needs a browser, create a dedicated window on that thread or use **WebViewCreateWindowEx** instead. Only one WebView instance can be attached to a given host window at a time. If you need to recreate the browser, call **WebViewDetachWindow** before attaching again.

The calling thread must be in a single-threaded apartment (STA). During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), the attach will fail. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

The dwOptions parameter enables you to specify what features are enabled in the browser. It is recommended that most applications use `WEBVIEW_OPTION_ENABLE_DEFAULT` which enables functionality which is most commonly found in web browsers. If you want your host application to exchange messages with scripts executing in the browser, you should also specify the `WEBVIEW_OPTION_ENABLE_MESSAGES` options. This option must be enabled to use **WebViewPostMessage** and **WebViewGetMessage** functions.

When clicking on a link would normally open that page in a new tab or browser window, it will be redirected to the primary browser window by default. This ensures that the browser control retains focus and all events are within the context of that window. If you wish to permit the creation of popup windows and/or support the creation of new browser windows (for example, if a link target is "_blank") then specify the `WEBVIEW_OPTION_ENABLE_NEW_WINDOWS` option. Note that if you permit the creation of new popup windows, it can shift focus away from your main browser window. There is no mechanism in the API to enumerate the child popup windows and they are independent of the window attached to the browser.

By default, the browser will use the language associated with the current user's locale. If you specify an alternate language, this also affects what languages the browser includes with the **Accept-Language** request header. This will provide hints to the website as to what language is preferred when it returns content to the browser. However, it does not guarantee that a website will return content in that language and will not automatically cause a page to be translated from one language to another.

If you pass NULL or an empty string for the ***lpszDataFolder*** parameter, the library creates a temporary, per-session user-data folder under the current user's `LocalAppData` folder. That folder is flagged as temporary and is automatically deleted when you detach the WebView from the window via **WebViewDetachWindow**. If you pass a relative path (such as "MyApp"), the library creates or

uses that folder beneath `%LocalAppData%\WebView`, creating a folder path like `%LocalAppData%\WebView\MyApp`.

If you pass a fully qualified path as the *lpszDataFolder* parameter, it will use that path. However, it will not accept paths to unsafe locations and will return an error if your application does not have permission to create a folder in that location. If there are environment variables surrounded by percent (%) symbols, those variables will be expanded when the folder path is resolved. If you enable privacy mode with `WEBVIEW_OPTION_ENABLE_PRIVACY`, the library will ensure the profile is unique to the session by appending a GUID to the base folder and treating it as temporary. The **WebViewGetUserDataFolder** function can be used to obtain the full path to the data folder being used.

When multiple WebView instances point at the same user-data folder, they share a single WebView2 profile created with that folder path. In this configuration the browser cache, cookies, history, and site storage are persisted under that folder and are reused across windows and processes, which speeds up subsequent navigations and preserves sign-in/session state. The library passes your folder directly to WebView2 when creating the environment, so any instance using that same path participates in the same profile, and you can clear portions of that profile at runtime via the provided helpers for cache, history, and cookies.

Your `WEBVIEWEVENTPROC` event callback function is invoked on the same thread that owns the WebView window. Because events are raised on the UI thread, long-running work in an event handler can block the user interface and stall navigation. Where possible, defer heavy operations to background threads. You should avoid displaying modal dialogs or message boxes from within event handlers. Also note that some events may fire multiple times during a single navigation. For example, the security state can transition through several levels (unknown, neutral, secure, or warning) before settling on a final value. Applications should be written to handle these changes incrementally.

To present correctly inside a resizable child window, resize the browser view when the host window size changes. The library tracks the host window position but relies on your WM_SIZE handling to drive layout.

Example

```
#include "cstools12.h"

// Windows message handler
static LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// WebView event handler for navigation and security events
static VOID CALLBACK WebViewEvents(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    switch (uEvent)
    {
    case WEBVIEW_EVENT_NAVIGATION_STARTING:
        OutputDebugString(_T("[WebView] Navigation starting\n"));
        break;

    case WEBVIEW_EVENT_NAVIGATION_COMPLETED:
        if (wParam)
            OutputDebugString(_T("[WebView] Navigation completed\n"));
        else
            OutputDebugString(_T("[WebView] Navigation failed\n"));
        break;

    case WEBVIEW_EVENT_SECURITY_CHANGED:
```

```
        OutputDebugString(_T("[WebView] Security changed\n"));
        break;
    }
}

INT WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR
lpCmdLine, INT nCmdShow)
{
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        MessageBox(NULL, _T("Unable to initialize the WebView API"), _T("Error"),
MB_ICONEXCLAMATION);
        return 0;
    }

    LPCTSTR kClass = _T("MyWebViewWndClass");
    WNDCLASS wc{};
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszClassName = kClass;
    RegisterClass(&wc);

    HWND hWnd = CreateWindowEx(0, kClass, _T("WebView"),
                               WS_OVERLAPPEDWINDOW,
                               CW_USEDEFAULT, CW_USEDEFAULT, 1024, 720,
                               NULL, NULL, hInstance, NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // A standard Windows message loop
    MSG uMsg;

    while (GetMessage(&uMsg, nullptr, 0, 0))
    {
        TranslateMessage(&uMsg);
        DispatchMessage(&uMsg);
    }

    WebViewUninitialize();
    return (int)uMsg.wParam;
}

static LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
        // Create a WebView2 browser instance and attach to this window
        {
            DWORD dwOptions = WEBVIEW_OPTION_ENABLE_DEFAULT;
            LPCTSTR pszInitialUrl = _T("https://example.com");
            LPCTSTR pszDataFolder = _T("MyTestApp");

            if (!WebViewAttachWindowEx(hWnd, dwOptions, pszInitialUrl, NULL,
pszDataFolder, WebViewEvents, 0))
```

```
            {
                MessageBox(NULL, _T("Unable to attach the WebView2 control"),
_T("Error"), MB_ICONEXCLAMATION);
            }
        }
        return 0;

    case WM_SIZE:
        // Make sure the browser window resizes with the window
        {
            RECT rc;
            GetClientRect(hWnd, &rc);
            WebViewSetBounds(hWnd, rc.left, rc.top, rc.right - rc.left, rc.bottom -
rc.top);
        }
        return 0;

    case WM_DESTROY:
        // Detach the WebView2 browser instance from the window
        WebViewDetachWindow(hWnd);
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewCreateWindowEx, WebViewDetachWindow, WebViewEventProc, WebViewGetUserDataFolder, WebViewInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewCancelDownload Function

```
BOOL WINAPI WebViewCancelDownload(
    HWND hWndHost,
    DWORD dwFileId
);
```

The **WebViewCancelDownload** function cancels the specified download in progress.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwFileId

An integer which specifies a unique file identifier. This value cannot be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewCancelDownload** function causes the specified file download to be canceled. If the transfer has completed or has already been canceled, this function has no effect. This function does not wait for the download to be canceled before it returns; it returns immediately after asynchronously requesting the download be canceled. The `WEBVIEW_EVENT_DOWNLOAD_COMPLETED` event will fire once the download has stopped.

The *dwFileId* value is obtained when the `WEBVIEW_EVENT_DOWNLOAD_STARTING` event is fired and the *lParam* event argument will contain the unique file identifier for the download. You should treat the identifier as an opaque integer and not depend on the value being sequential. It is only guaranteed that the value will be unique for all downloads during the session and is only valid for the browser that initiated the download. Note there can be multiple downloads in progress simultaneously.

The **WebViewGetDownloadInfo** function can be used to get the status of the download and determine if it completed successfully, failed or was canceled.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewCancelNavigation, WebViewEnumDownloads, WebViewGetDownloadInfo, WebViewGetDownloadPath, WebViewSetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewCancelNavigation Function

```
BOOL WINAPI WebViewCancelNavigation(
     HWND hWndHost
);
```

The **WebViewCancelNavigation** function cancels any current or pending navigations.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function can be safely called at any time, but only has an effect if the browser is currently in the process of navigating to a new location. A canceled navigation will fire the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event and the event handler's *wParam* value will be FALSE. The navigation status will also include the `WEBVIEW_STATUS_NAVIGATION_CANCELED` bitflag which can be used to distinguish between a navigation that was canceled and one that failed for another reason.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetNavigationStatus, WebViewNavigateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewClearCache Function

```
BOOL WINAPI WebViewClearCache(
    HWND hWndHost
);
```

The **WebViewClearCache** function removes visited pages from the cache.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function will clear the browser cache for the current session. This can be useful if you wish to force the browser to reload content. If you have created the browser instance using a temporary data folder, or you have specified the `WEBVIEW_OPTION_ENABLE_PRIVACY` option, the browser session will always start with a new user profile and will not contain any cached content.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCookies, WebViewClearHeaders, WebViewClearHistory, WebViewRedrawWindow, WebViewReloadDocument

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewClearCookies Function

```
BOOL WINAPI WebViewClearCookies(
    HWND hWndHost,
    BOOL bAllCookies
);
```

The **WebViewClearCookies** function deletes cookies for the current profile or session.

Parameters

hWndHost

Handle to the window attached to the browser session.

bAllCookies

A boolean value which specifies if all cookies for the current user profile should be deleted, or only those cookies created during the current session. If this parameter is a non-zero value then all cookies stored in the current user's profile will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function deletes cookies that have been created by the current WebView profile. If the *bAllCookies* parameter is set to TRUE, all stored cookies for every site are removed; if set to FALSE, only the cookies associated with the current session are cleared. The effect is comparable to clearing cookies in a regular browser: login sessions, preferences, and other state information stored in cookies will no longer be available. This only affects the WebView's private user data folder and does not modify cookies stored by the user's standalone Edge browser.

If you have used the WebView browser to authenticate with one or more websites, calling this function will typically log you out of those sites and require you to re-authenticate. If you have created the browser instance using a temporary data folder, or you have specified the `WEBVIEW_OPTION_ENABLE_PRIVACY` option, no cookies will be preserved after the session ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCache, WebViewClearHeaders, WebViewClearHistory

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewClearHeaders Function

```
BOOL WINAPI WebViewClearHeaders(
    HWND hWndHost
    DWORD dwHeaderType
);
```

The **WebViewClearHeaders** function clears the current request and/or response headers for the current session.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwHeaderType

An unsigned integer value which may be one of the following values.

Value	Description
WEBVIEW_HEADERS_REQUEST	The custom request headers for the current session should be deleted. These are the HTTP headers which can be set or changed prior to the browser navigating to a new page. The standard request headers included with every request will continue to be sent.
WEBVIEW_HEADERS_RESPONSE	The response headers for the current session should be deleted. These are the headers which are returned by the server after navigation has completed. These headers are automatically deleted with each new navigation, making this option unnecessary in most cases.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function can be used to delete the custom request headers set with **WebViewSetHeader** or clear the response headers returned by the previous navigation.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCache, WebViewClearCookies, WebViewClearHistory, WebViewGetHeader, WebViewSetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewClearHistory Function

```
BOOL WINAPI WebViewClearHistory(
    HWND hWndHost
);
```

The **WebViewClearHistory** function clears the navigation history for the current session.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function clears both the profile's stored browsing history and the navigation history for the current WebView instance. Internally, it uses the WebView2 profile APIs to remove stored history and related download entries, and then resets the back/forward navigation stack. The effect is similar to clearing the browser history in a browser, where visited pages are removed from the user's history, and the in-memory list of sites you can go back or forward to is discarded.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCache, WebViewClearCookies, WebViewClearHeaders, WebViewGoBack, WebViewGoForward

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewCreateWindow Function

```
HWND WINAPI WebViewCreateWindow(
   LPCTSTR lpszUrl,
   WEBVIEWEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **WebViewCreateWindow** function creates a new window which hosts an instance of the browser control.

Parameters

lpszUrl

A null-terminated string which specifies the initial URL the browser should navigate to after the window has been created. If this parameter is NULL or an empty string, the browser instance will be created with a blank page. The only schemes permitted are `https://`, `http://` and `file://`. If a domain name is specified without a scheme, it will default to using `https://`.

lpEventProc

A callback function which will be invoked whenever browser events occur. If the application does not need to process events, this parameter can be NULL. The callback function will be invoked in the context of a separate thread which hosts the browser control, not the thread which calls this function. For more information, refer to the WebViewEventProc and WebView Events documentation.

dwParam

An optional parameter which can be used to provide additional context to the event handler. This user-defined value is passed back to the callback function when an event occurs. If this information is not required, this parameter can be zero.

Return Value

If the function succeeds, the return value is the handle of the window which contains the browser control. If the function fails, the return value is NULL. To get extended error information, call **WebViewGetLastError**.

Remarks

This function creates a top-level host window and a WebView runtime on a dedicated UI thread owned by the library. It returns after the window and control have been created and any navigation to the initial URL proceeds asynchronously. The returned window handle can be managed with normal Win32 APIs (such as **ShowWindow**), but you should destroy it with **WebViewDestroyWindow** rather than calling **DestroyWindow** directly. This ensures the browser resources are released correctly. If you specify an event handler, it will receive the normal attach and detach events associated with creating a browser session.

This function requires the single-threaded apartment (STA) COM model. During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), a browser instance cannot be created. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

Event callbacks are invoked on the thread that owns the browser window, not the thread that calls **WebViewCreateWindow**. This is typically the internal UI thread that runs the message loop for the control. Avoid updating shared globals from both your main thread and the browser thread without

synchronization. If you need to communicate back to your main UI, use **PostMessage** or **SendMessage** to send application-defined messages to the window, or use an event, critical section, or message queue rather than directly changing the shared state.

By default the browser instance is created with WEBVIEW_OPTION_ENABLE_DEFAULT, enabling common features such as accelerator keys, the context menu, status text, zoom control, audio playback, JavaScript execution, and script dialogs. This gives a familiar, browser-like experience without additional configuration. If you need a more locked-down view (for kiosk or embedded scenarios), use **WebViewCreateWindowEx** to specify different options.

The function can be used from a console application. You do not need to create a message pump on your main thread; the library's hosting thread runs the necessary loop for the browser window. GUI applications can also call this function to create a modeless browser window that you control like any other top-level window.

The window title is automatically updated to reflect the title of the current page as navigation completes.

Example

```
#include "cstools12.h"
#include <stdio.h>

// WebView event handler for navigation and security events
static VOID CALLBACK WebViewEvents(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    switch (uEvent)
    {
    case WEBVIEW_EVENT_NAVIGATION_STARTING:
        OutputDebugString(_T("[WebView] Navigation starting\n"));
        break;

    case WEBVIEW_EVENT_NAVIGATION_COMPLETED:
        if (wParam)
            OutputDebugString(_T("[WebView] Navigation completed\n"));
        else
            OutputDebugString(_T("[WebView] Navigation failed\n"));
        break;

    case WEBVIEW_EVENT_SECURITY_CHANGED:
        OutputDebugString(_T("[WebView] Security changed\n"));
        break;
    }
}

int _tmain(int argc, TCHAR *argv[])
{
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Unable to initialize the WebView API, error 0x%08lx\n"),
WebViewGetLastError());
        return 1;
    }

    // Create a top-level browser window
    HWND hBrowser = WebViewCreateWindow(_T("https://example.com"), WebViewEvents,
0);
```

```
    if (!hBrowser)
    {
        _tprintf(_T("Unable to create a WebView window\n"));
        return 2;
    }

    TCHAR szCmdLine[1024];
    TCHAR *pszCmd;
    TCHAR *pszArg;

    // Read commands from standard input to control the browser
    while (pszCmd = _getts_s(szCmdLine, _countof(szCmdLine)))
    {
        // Trim any leading whitespace
        while (*pszCmd && _istspace(*pszCmd))
            pszCmd++;

        // Look for any arguments to the command
        pszArg = _tcschr(pszCmd, _T(' '));
        if (pszArg)
            *pszArg++ = TCHAR('\0');

        if (_tcsicmp(pszCmd, _T("url")) == 0)
        {
            // The URL command will navigate to a new URL
            if (pszArg && _tcslen(pszArg) > 0)
            {
                if (!WebViewNavigateUrl(hBrowser, WEBVIEW_NAVIGATE_DEFAULT,
pszArg))
                {
                     DWORD dwError = WebViewGetLastError();
                     _tprintf(_T("WebViewNavigateUrl failed with error 0x%08lx\n"),
dwError);
                }
            }
            else
            {
                _tprintf(_T("This command requires a URL\n"));
            }
        }
        else if (_tcsicmp(pszCmd, "quit") == 0)
        {
            // Exit the loop, closing the browser window
            break;
        }
    }

    // Destroy the browser window and wait for it to close
    WebViewDestroyWindow(hBrowser);
    WebViewWaitUntilClosed(hBrowser, INFINITE);

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindow, WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewDestroyWindow, WebViewEventProc, WebViewInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewCreateWindowEx Function

```
HWND WINAPI WebViewCreateWindowEx(
   LPCTSTR lpszTitle,
   DWORD dwStyle,
   LPRECT lpPosition,
   DWORD dwOptions,
   LPCTSTR lpszUrl,
   LPCTSTR lpszLanguage,
   LPCTSTR lpszDataFolder,
   WEBVIEWEVENTPROC lpEventProc,
   DWORD dwParam
);
```

The **WebViewCreateWindowEx** function creates a new window which hosts an instance of the browser control. This extended version of the function allows the developer to provide additional options, an alternate display language and specify an optional folder where the browser will store session data.

Parameters

lpszTitle

A null-terminated string which specifies the text displayed in the title bar. This parameter may be NULL or an empty string, in which case the title bar is set to the title of the document currently being displayed in the browser and will be automatically updated with subsequent navigations.

dwStyle

An unsigned integer which specifies the window style. This is the same windows style used with the **CreateWindowEx** function. If the value is zero, it will default to using the `WS_OVERLAPPEDWINDOW` style which creates an overlapped window with a caption, system menu and a minimize and maximize button. Some style bitflags will be ignored, such as the `WS_CHILD` style which cannot be applied to this window. This function can only be used to create an application window.

lpPosition

A pointer to a **RECT** structure which defines the position and size of the window. If all structure member values are `>= 0`, their values are passed directly to **CreateWindowEx**. If the top or left values are -1, a default position will be used. For example, if the structure is defined as `{ -1, -1, 800, 600 }` it would create an 800 x 600 window at a default position on the display. If the left or right values are -1, default values are used for the width or height. If all values are negative, the position and size of the window will use default values. If this parameter is NULL, the window will be positioned in the center of the current display with a width and height set to half the display area.

dwOptions

One or more option bitflags. For more information, refer to the WebView Options documentation. Most applications should not use a value of zero for this parameter. If no options are specified, the browser instance will be created with a very restrictive set of permissions that can prevent many websites from functioning correctly.

lpszUrl

A null-terminated string which specifies the initial URL the browser should navigate to after the window has been created. If this parameter is NULL or an empty string, the browser instance will be created with a blank page. The only schemes permitted are `https://`, `http://` and `file://`. If a domain name is specified without a scheme, it will default to using `https://`.

lpszLanguage

A null-terminated string which specifies an alternate language to use when the browser displays a context menu or dialogs. This value can be a standard language tag in BCP-47 format (such as "en-US") or it can be a string which specifies a hexadecimal Windows locale identifier (such as "0x0409"). It is recommended you use the language tag whenever possible, as Microsoft is deprecating the use of locale identifiers (LCIDs) and their associated APIs. If this parameter is a NULL pointer or an empty string, the browser will use the current default language for the local user.

lpszDataFolder

A folder name where the user profile data will be stored. This is the location where the browser will cache viewed pages, store cookies and form data, and where scripts may store session data. This value may be just a name (such as the name of the application) or an absolute path. If the folder name contains environment variables surrounded in percent (%) symbols, they will be replaced with their value in the current user's environment. If folder name does specify an absolute path, the profile will be stored in the current user's LocalAppData folder. If this parameter is NULL or an empty string, a temporary profile will be created for the session and then automatically deleted when the session ends.

lpEventProc

A callback function which will be invoked whenever browser events occur. If the application does not need to process events, this parameter can be NULL. The callback function will be invoked in the context of a separate thread which hosts the browser control, not the thread which calls this function. For more information, refer to the WebViewEventProc and WebView Events documentation.

dwParam

An optional parameter which can be used to provide additional context to the event handler. This user-defined value is passed back to the callback function when an event occurs. If this information is not required, this parameter can be zero.

Return Value

If the function succeeds, the return value is the handle of the window which contains the browser control. If the function fails, the return value is NULL. To get extended error information, call **WebViewGetLastError**.

Remarks

This function creates a top-level host window and a WebView runtime on a dedicated UI thread owned by the library. It returns after the window and control have been created and any navigation to the initial URL proceeds asynchronously. The returned window handle can be managed with normal Win32 APIs (such as **ShowWindow**), but you should destroy it with **WebViewDestroyWindow** rather than calling **DestroyWindow** directly. This ensures the browser resources are released correctly. If you specify an event handler, it will receive the normal attach and detach events associated with creating a browser session.

This function requires the single-threaded apartment (STA) COM model. During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), a browser instance cannot be created. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

The dwOptions parameter enables you to specify what features are enabled in the browser. It is recommended that most applications use `WEBVIEW_OPTION_ENABLE_DEFAULT` which enables functionality which is most commonly found in web browsers. If you want your host application to exchange messages with scripts executing in the browser, you should also specify the

`WEBVIEW_OPTION_ENABLE_MESSAGES` options. This option must be enabled to use **WebViewPostMessage** and **WebViewGetMessage** functions.

When clicking on a link would normally open that page in a new tab or browser window, it will be redirected to the primary browser window by default. This ensures that the browser control retains focus and all events are within the context of that window. If you wish to permit the creation of popup windows and/or support the creation of new browser windows (for example, if a link target is "_blank") then specify the `WEBVIEW_OPTION_ENABLE_NEW_WINDOWS` option. Note that if you permit the creation of new popup windows, it can shift focus away from your main browser window. There is no mechanism in the API to enumerate the child popup windows and they are independent of the window you have created.

By default, the browser will use the language associated with the current user's locale. If you specify an alternate language, this also affects what languages the browser includes with the **Accept-Language** request header. This will provide hints to the website as to what language is preferred when it returns content to the browser. However, it does not guarantee that a website will return content in that language and will not automatically cause a page to be translated from one language to another.

If you pass NULL or an empty string for the *lpszDataFolder* parameter, the library creates a temporary, per-session user-data folder under the current user's `LocalAppData` folder. That folder is flagged as temporary and is automatically deleted when you detach the WebView from the window via **WebViewDetachWindow**. If you pass a relative path (such as "MyApp"), the library creates or uses that folder beneath `%LocalAppData%\WebView`, creating a folder path like `%LocalAppData%\WebView\MyApp`.

If you pass a fully qualified path as the *lpszDataFolder* parameter, it will use that path. However, it will not accept paths to unsafe locations and will return an error if your application does not have permission to create a folder in that location. If there are environment variables surrounded by percent (%) symbols, those variables will be expanded when the folder path is resolved. If you enable privacy mode with `WEBVIEW_OPTION_ENABLE_PRIVACY`, the library will ensure the profile is unique to the session by appending a GUID to the base folder and treating it as temporary. The **WebViewGetUserDataFolder** function can be used to obtain the full path to the data folder being used.

When multiple WebView instances point at the same user-data folder, they share a single WebView2 profile created with that folder path. In this configuration the browser cache, cookies, history, and site storage are persisted under that folder and are reused across windows and processes, which speeds up subsequent navigations and preserves sign-in/session state. The library passes your folder directly to WebView2 when creating the environment, so any instance using that same path participates in the same profile, and you can clear portions of that profile at runtime via the provided helpers for cache, history, and cookies.

Event callbacks are invoked on the thread that owns the browser window, not the thread that calls **WebViewCreateWindowEx**. This is typically the internal UI thread that runs the message loop for the control. Avoid updating shared globals from both your main thread and the browser thread without synchronization. If you need to communicate back to your main UI, use **PostMessage** or **SendMessage** to send application-defined messages to the window, or use an event, critical section, or message queue rather than directly changing the shared state.

The function can be used from a console application. You do not need to create a message pump on your main thread; the library's hosting thread runs the necessary loop for the browser window. GUI applications can also call this function to create a modeless browser window that you control like any other top-level window.

Example

```
#include "cstools12.h"
#include <stdio.h>

// WebView event handler for navigation and security events
static VOID CALLBACK WebViewEvents(HWND hWnd, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    switch (uEvent)
    {
    case WEBVIEW_EVENT_NAVIGATION_STARTING:
        OutputDebugString(_T("[WebView] Navigation starting\n"));
        break;

    case WEBVIEW_EVENT_NAVIGATION_COMPLETED:
        if (wParam)
            OutputDebugString(_T("[WebView] Navigation completed\n"));
        else
            OutputDebugString(_T("[WebView] Navigation failed\n"));
        break;

    case WEBVIEW_EVENT_SECURITY_CHANGED:
        OutputDebugString(_T("[WebView] Security changed\n"));
        break;
    }
}

int _tmain(int argc, TCHAR *argv[])
{
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        _tprintf(_T("Unable to initialize the WebView API, error 0x%08lx\n"),
WebViewGetLastError());
        return 1;
    }

    // Create a top-level browser window
    LPCTSTR pszAppTitle = _T("MyWebView");
    LPCTSTR pszInitialUrl = _T("https://example.com");
    LPCTSTR pszDataFolder = _T("MyTestApp");
    DWORD dwOptions = WEBVIEW_OPTION_ENABLE_DEFAULT;
    RECT rcPos = { -1, -1, 1200, 800 };

    HWND hBrowser = WebViewCreateWindowEx(
                            pszAppTitle,          // Window title
                            WS_OVERLAPPEDWINDOW,  // Window style
                            &rcPos,               // Initial position
                            dwOptions,            // Creation options
                            pszInitialUrl,        // Initial URL
                            NULL,                 // Default language
                            pszDataFolder,        // User data folder
                            WebViewEvents, 0);    // Event callback
    if (!hBrowser)
    {
        _tprintf(_T("Unable to create a WebView window\n"));
        return 2;
    }
```

```
    TCHAR szCmdLine[1024];
    TCHAR *pszCmd;
    TCHAR *pszArg;

    // Read commands from standard input to control the browser
    while (pszCmd = _getts_s(szCmdLine, _countof(szCmdLine)))
    {
        // Trim any leading whitespace
        while (*pszCmd && _istspace(*pszCmd))
            pszCmd++;

        // Look for any arguments to the command
        pszArg = _tcschr(pszCmd, _T(' '));
        if (pszArg)
            *pszArg++ = TCHAR('\0');

        if (_tcsicmp(pszCmd, _T("url")) == 0)
        {
            // The URL command will navigate to a new URL
            if (pszArg && _tcslen(pszArg) > 0)
            {
                if (!WebViewNavigateUrl(hBrowser, WEBVIEW_NAVIGATE_DEFAULT,
pszArg))
                {
                     DWORD dwError = WebViewGetLastError();
                     _tprintf(_T("WebViewNavigateUrl failed with error 0x%08lx\n"),
dwError);
                }
            }
            else
            {
                _tprintf(_T("This command requires a URL\n"));
            }
        }
        else if (_tcsicmp(pszCmd, "quit") == 0)
        {
            // Exit the loop, closing the browser window
            break;
        }
    }

    // Destroy the browser window and wait for it to close
    WebViewDestroyWindow(hBrowser);
    WebViewWaitUntilClosed(hBrowser, INFINITE);

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindow, WebViewAttachWindowEx, WebViewCreateWindow,

WebViewDestroyWindow, WebViewEventProc, WebViewGetUserDataFolder, WebViewInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewDestroyWindow Function

```
BOOL WINAPI WebViewDestroyWindow(
    HWND hWndHost
);
```

The **WebViewDestroyWindow** function closes the window and releases all browser resources.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function closes a browser window which was created using the **WebViewCreateWindow** or **WebViewCreateWindowEx** function. It is recommended that you always use this function to close the window rather than posting a **WM_CLOSE** message because it ensures resources for the window, such as its icons, are released correctly. The process of destroying the window does not occur immediately and when this function returns it is not guaranteed the browser child window and its associated processes have completely stopped. Use the **WebViewWaitUntilClosed** function if you wish to wait for the cleanup process to complete.

This function will cause the **WEBVIEW_EVENT_WINDOW_DETACHED** event to fire as part of the normal process of destroying the window. If you implement a handler for this event, you should never delay this process by attempting to show a dialog or request input from the user (for example, by displaying a message box). At the point this event is fired, the browser objects have already been disposed and there is no way to prevent the process from completing.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewCreateWindow, WebViewCreateWindowEx, WebViewDetachWindow, WebViewWaitUntilClosed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewDetachWindow Function

```
BOOL WINAPI WebViewDetachWindow(
    HWND hWndHost
);
```

The **WebViewDetachWindow** function releases the resources allocated for the browser session.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function is used to detach the browser instance from a window that it was previously attached to using either the **WebViewAttachWindow** or **WebViewAttachWindowEx** functions. This function should always be called when the browser is no longer needed and the host window is being destroyed. If the window handle was created using the **WebViewCreateWindow** or **WebViewCreateWindowEx** function, calling this function has the same effect as calling the **WebViewDestroyWindow** function.

This function will cause the `WEBVIEW_EVENT_WINDOW_DETACHED` event to fire as part of the normal process of detaching from the window. If you implement a handler for this event, you should never delay this process by attempting to show a dialog or request input from the user (for example, by displaying a message box). At the point this event is fired, the browser objects have already been disposed and there is no way to prevent the process from completing.

If there is a navigation in progress at the time this function is called, it will be canceled. If there are one or more file downloads in progress, the transfers will be aborted.

The current thread will block while it waits for the WebView2 browser process to shutdown gracefully. If the browser becomes non-responsive, this function will wait several seconds before forcibly terminating the child process and continuing with the cleanup process. If a temporary user profile was specified or a private browsing session was created, this function will also clear the browser cache and remove the user data folder that was created for the session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewAttachWindowEx, WebViewDestroyWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewDisableEvents Function

```
BOOL WINAPI WebViewDisableEvents(
    HWND hWndHost
);
```

The **WebViewDisableEvents** function disables event handling for the specified browser.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function removes the currently installed event handler for the specified host window and clears the user parameter. After it returns, the application will not receive any further browser events until **WebViewEnableEvents** is called again with a valid handler. Use this to permanently stop participating in event notifications (for example, during shutdown or when detaching the browser control from the host). Unlike temporarily freezing events, this function completely clears the callback handler for the browser instance.

If you only need to pause notifications temporarily, such as when presenting a modal dialog, performing a batch of UI updates, or executing several programmatic navigations back-to-back, freezing events is the recommended approach. Disabling events should be reserved for longer-term or terminal states, typically when the application is in the process of closing down.

You should not call **WebViewDisableEvents** from inside an event handler. Disabling callbacks while you are actively processing an event can lead to subtle reentrancy issues and missed cleanup. If you need to suspend notifications from within an event, call **WebViewFreezeEvents** instead, complete your work, and then unfreeze when you are ready to resume event handling.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewEnableEvents, WebViewEventProc, WebViewFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewEnableEvents Function

```
BOOL WINAPI WebViewEnableEvents(
   HWND hWndHost,
   WEBVIEWEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **WebViewEnableEvents** function enables event handling for the specified browser.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpEventProc

A callback function which will be invoked whenever browser events occur. This parameter cannot be NULL. The callback function will be invoked in the context of a separate thread which hosts the browser control, not the thread which calls this function. For more information, refer to the WebViewEventProc and WebView Events documentation.

dwParam

An optional parameter which can be used to provide additional context to the event handler. This user-defined value is passed back to the callback function when an event occurs. If this information is not required, this parameter can be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewEnableEvents** function assigns or replaces the callback function that receives browser events for the specified host window. It can be used after the browser has been attached to start receiving notifications or to change event handlers at runtime. The change takes effect immediately and subsequent events will be invoked using the new handler.

Event delivery is serialized by the library and this call safely updates the event handler. You cannot specify a NULL pointer as the event handler; if you need to disable events, use the **WebViewDisableEvents** function. If events were previously frozen with **WebViewFreezeEvents**, calling this function with a new event handler will automatically unfreeze event notifications.

When using this function from within a C++ class, the event callback must be declared as a static function with the **__stdcall** (or **CALLBACK**) calling convention. This is required because the library calls the handler just as it would with any Win32 API callback function, without knowledge of C++ instance objects. Non-static member functions include a hidden **this** pointer parameter, which makes their signature incompatible with the expected **WEBVIEWEVENTPROC** type. A common pattern is to pass the object's ***this*** pointer as the ***dwParam*** value, then cast it back inside the static callback function to access the instance. This preserves object-oriented design while still meeting the calling convention requirements of the API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewDisableEvents, WebViewEventProc, WebViewFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewEnumDownloads Function

```
INT WINAPI WebViewEnumDownloads(
    HWND hWndHost,
    DWORD dwStatus,
    LPDWORD lpdwFileList,
    INT nMaxFiles
);
```

The **WebViewEnumDownloads** function enumerates the file identifiers for downloads during the current session.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwStatus

A value which specifies the status of the downloads to enumerate. If this value is zero, the function will enumerate all downloads. If this parameter is one of the values listed below, only those downloads which match the specified status will be enumerated. For example, if this parameter is `WEBVIEW_DOWNLOAD_COMPLETED` then only those file downloads which have completed will be enumerated.

Name	Description
WEBVIEW_DOWNLOAD_STARTED	The file download has started and is in-progress.
WEBVIEW_DOWNLOAD_COMPLETED	The file download has completed successfully.
WEBVIEW_DOWNLOAD_CANCELED	The file download has been canceled.
WEBVIEW_DOWNLOAD_FAILED	The file download failed due to a network error or other problem.

lpdwFileList

A pointer to an array of DWORD values which specify the unique file identifier associated with the download. The identifiers can be passed to the **WebViewGetDownloadInfo** function to get detailed information about a specific download. This parameter can be NULL, in which case it will be ignored and the function will return the number of downloaded files.

nMaxFiles

The maximum number of file identifiers which can be copied into the *lpdwFileList* array. If the *lpdwFileList* parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the number of files downloaded during the current session. If there have been no downloads, or if the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewEnumDownloads** function can be used to obtain the identifiers for one or more file downloads that have occurred during the current browser session. Each identifier returned by this function uniquely corresponds to a download and can be used with functions such as **WebViewGetDownloadInfo** or **WebViewGetDownloadPath** to obtain detailed information about

the file.

If the *dwStatus* parameter is set to zero, the function enumerates all downloads regardless of their state. Otherwise, only downloads that match the specified status value are included in the enumeration. This allows applications to filter the results to include only downloads that are currently in progress, those that completed successfully, or those that were canceled or failed.

When the *lpdwFileList* parameter is not NULL, the function copies up to *nMaxFiles* identifiers into the specified array. If the number of matching downloads exceeds the array size, only the first *nMaxFiles* identifiers are returned, but the return value will still reflect the total number of matching downloads. This enables the application to first call the function with a NULL pointer to determine how many downloads are available, and then allocate an appropriately sized buffer before retrieving the identifiers.

The enumeration includes both active and completed downloads. If the host window is detached or destroyed, the list of downloads is cleared and further enumeration will return zero results.

Example

```
// Get the total number of downloads
INT nTotal = WebViewEnumDownloads(hWndHost, 0, NULL, 0);

if (nTotal <= 0)
{
    _tprintf(_T("No downloads found.\n"));
    return;
}

// Allocate a buffer for the identifiers
DWORD *pIds = (DWORD*)malloc(nTotal * sizeof(DWORD));
if (!pIds)
    return;

INT nCount = WebViewEnumDownloads(hWndHost, 0, pIds, nTotal);

// Enumerate each download and print basic info
for (INT nIndex = 0; nIndex < nCount; nIndex++)
{
    WEBVIEW_DOWNLOAD_INFO fileInfo = { sizeof(fileInfo) };

    if (WebViewGetDownloadInfo(hWndHost, pIds[nIndex], &fileInfo))
    {
        _tprintf(_T("Download %u: %s (%llu/%llu bytes)\n"),
                 fileInfo.dwFileId,
                 fileInfo.szFileName,
                 fileInfo.uiBytesCopied.QuadPart,
                 fileInfo.uiBytesTotal.QuadPart);
    }
}

free(pIds);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewCancelDownload, WebViewGetDownloadInfo, WebViewGetDownloadPath, WebViewSetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewEventProc Function

```
VOID CALLBACK WebViewEventProc(
    HWND hWndHost,
    UINT uEventId,
    WPARAM wParam,
    LPARAM lParam,
    DWORD_PTR dwParam
);
```

The **WebViewEventProc** function is an application-defined callback function that processes events generated by the browser.

Parameters

hWndHost

Handle to the window attached to the browser session.

uEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the WebView Events documentation.

wParam

An integer value that depends on the event context. This value is commonly used by an event to indicate success (1) or failure (0). It may also be used to indicate a transition in state. Refer to the events documentation for information about how this parameter may be used by a specific event.

lParam

An integer value that depends on the event context. This value is commonly used by an event to provide a status code or error code, depending whether the operation which raised the event succeeded or failed. Refer to the events documentation for information about how this parameter may be used by a specific event.

dwParam

A user-defined integer value which was provided when the event callback was specified. This is typically used to provide additional context to the event or associate the event with a specific object. If this user-defined parameter was not specified when the browser window was attached or created, this value will be zero.

Return Value

None.

Remarks

The library invokes the callback function whenever an event occurs. Applications are not required to implement event handlers for each event, and may choose to not provide any event handlers by passing NULL as the callback function parameter. A minimal implementation would typically create handlers for the `WEBVIEW_EVENT_NAVIGATION_STARTING` event which notifies the application when the browser is beginning to navigate to a new location, and the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event which indicates the navigation has completed.

When you attach a browser instance to an existing window, the callback function is always invoked on the same UI thread that owns that window, ensuring events are processed in the context of the caller's thread. However, if you create a new browser window using the **WebViewCreateWindow** function, a different thread manages the top-level window and message loop, and in that case the callback

function is invoked on the thread that owns the browser window rather than the thread that created it. This distinction is important when your application accesses shared data or resources from within the event handler. To avoid race conditions or undefined behavior, ensure that access to shared state is synchronized using mutexes, critical sections, or other appropriate synchronization primitives.

It is important to keep event handlers responsive. Because events are raised on the UI thread, long-running work in an event handler can block the user interface and stall navigation. Where possible, defer heavy operations to background threads. Avoid displaying modal dialogs or message boxes from within event handlers. Also note that some events may fire multiple times during a single navigation. For example, the security state can transition through several levels (unknown, neutral, secure, or warning) before settling on a final value. Applications should be written to handle these changes incrementally.

After the browser has been attached to a window, it is possible to change the event handler using the **WebViewEnableEvents** function. This changes the event handler for the browser instance, and from that point forward all event notifications will be routed to the new handler.

If you are implementing the callback function in a language other than C/C++ it must use the **__stdcall** calling convention, not the **__cdecl** calling convention. The parameters **WPARAM**, **LPARAM**, and **DWORD_PTR** used with the event callback are defined by the Windows API as integer types whose size depends on the target platform. For 32-bit applications, these types are 32-bit values; for 64-bit applications, they expand to 64-bit values. This design allows the parameters to hold either numeric values or pointer values in a way that is natural for the platform.

For applications written in C or C++, the compiler automatically adjusts these types based on whether you are building a 32-bit or 64-bit target. However, when calling the API from other languages (such as C#, Delphi, or VB), it is important to define these parameters as pointer-sized integers rather than fixed 32-bit integers. In .NET, for example, the correct type is **IntPtr**, which automatically maps to 32 or 64 bits depending on the runtime. In Delphi, you should use **NativeInt** or **NativeUInt**. Using a fixed 32-bit type in a 64-bit build can lead to truncated values or incorrect pointer conversions, while using a 64-bit type in a 32-bit build can cause calling convention mismatches.

When defining an event handler within a C++ class, the event callback must be declared as a static function with the **__stdcall** calling convention. This is required because the library calls the handler just as it would with any Win32 API callback function, without knowledge of C++ instance objects. Non-static member functions include a hidden **this** pointer parameter, which makes their signature incompatible with the expected **WEBVIEWEVENTPROC** type. A common pattern is to pass the object's ***this*** pointer as the ***dwParam*** value, then cast it back inside the static callback function to access the instance. This preserves object-oriented design while still meeting the calling convention requirements of the API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewAttachWindowEx, WebViewCreateWindow, WebViewCreateWindowEx, WebViewDisableEvents, WebViewEnableEvents, WebViewFreezeEvents, WebViewInEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewExecute Function

```
INT WINAPI WebViewExecute(
   HWND hWndHost,
   LPCTSTR lpszJavaScript,
   LPTSTR lpszResult,
   INT nMaxLength
);
```

The **WebViewExecute** function executes JavaScript and returns the result synchronously.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszJavaScript

A null-terminated string which specifies the JavaScript that should be executed in the context of the current page being viewed. This parameter is required and cannot be NULL.

lpszResult

A string buffer which will contain the result returned by the script, either as a JSON object or a JSON string value. If this parameter is NULL, the function will not copy the result but will return the total number of characters in the response, not including the terminating null.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null. If the ***lpszResult*** parameter is NULL, this parameter should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the result buffer, not including the terminating null. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewExecute** function evaluates a JavaScript expression within the context of the currently loaded page and returns the result synchronously. Unlike **WebViewAsyncExecute**, this function blocks the current thread while waiting for the script to complete. If the web page is non-responsive, this function will timeout after approximately two seconds.

If you wish to dynamically allocate a string buffer for the response, you can call this function with the ***lpszResult*** parameter as NULL and the ***nMaxLength*** parameter as zero. The function will return the number of characters in the response, not including the terminating null. You can allocate the buffer and then call **WebViewGetResult** which will copy the response into the buffer you provide. Because script execution is dispatched to the underlying WebView2 control, this function will fail if navigation is still in progress. Applications should wait for the `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event before invoking script execution.

Example

```
// Execute a simple script that returns a string
INT nLength = WebViewExecute(hWndHost, _T("document.title"), NULL, 0);

if (nLength > 0)
{
```

```
    // Allocate a buffer large enough to hold the result
    LPTSTR lpszResult = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (lpszResult)
    {
        // Retrieve the result of the script execution
        if (WebViewGetResult(hWndHost, lpszResult, nLength + 1))
        {
            _tprintf(_T("Page title is: %s\n"), lpszResult);
        }
        free(lpszResult);
    }
}
else
{
    _tprintf(_T("Script execution failed\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAsyncExecute, WebViewGetResult

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewFindWindow Function

```
HWND WINAPI WebViewFindWindow(
    HWND hWndParent,
    LPCTSTR lpszSearch,
    DWORD dwFlags,
    LPDWORD lpdwThreadId
);
```

The **WebViewFindWindow** function locates the associated host or child window handle for a browser instance.

Parameters

hWndParent

Handle to the parent window. If this parameter is NULL it will search all top-level windows. If this parameter is not NULL, it must be a valid window handle and the function will only match against a window which is a child window of the specified handle.

lpszSearch

An optional null-terminated string which specifies a URL. The function will only match those browser windows which have navigated to the specified URL. A simple case-insensitive substring search is performed, but will only match windows created by the current process. If this parameter is NULL or an empty string, the function will not match against any specific URL.

dwFlags

This parameter is reserved for future use and should always be zero. Future versions may expand this to support additional types of matching, such as document titles or more advanced search expressions.

lpdwThreadId

An optional parameter which is a pointer to an unsigned integer value that will contain the thread ID for the browser instance if a matching window is found. If no browser is found, this value will be set to zero. If this information is not required, this parameter can be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewFindWindow** function can be used to locate the window handle for an existing WebView2 instance that was created by the calling process. This is useful if the application needs to re-establish a reference to a previously created browser window, for example when handling messages or coordinating multiple windows.

If a parent window handle is specified, the function will restrict its search to that window and its children. This is the most efficient approach when the caller already knows the expected parent container. If the parameter is NULL, all top-level windows owned by the process are searched. An optional URL string can also be provided to limit results to windows whose current location contains the specified substring. The search is case-insensitive and only applies to windows created by the current process.

The function only returns a single matching window handle. If multiple instances could match the criteria, the particular instance returned is not guaranteed and the caller should design accordingly.

When both the parent handle and search string are NULL, the function simply locates the first available WebView2 instance owned by the current thread.

If the optional ***lpdwThreadId*** parameter is supplied, it will receive the thread identifier that owns the WebView2 instance. This can be used when thread affinity is important, such as posting messages directly to the UI thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewCreateWindow, WebViewCreateWindowEx, WebViewGetChildWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewFreezeEvents Function

```
BOOL WINAPI WebViewFreezeEvents(
    HWND hWndHost,
    BOOL bFreezeEvents
);
```

The **WebViewEnableEvents** function suspends or re-enables event notifications for the specified browser.

Parameters

hWndHost

Handle to the window attached to the browser session.

bFreezeEvents

An integer value which specifies if event handling should be frozen. If the value is non-zero, browser events will be suspended until they are re-enabled. If the value is zero, browser events will be re-enabled if an event handler has been defined.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function toggles a per-instance switch for event delivery. When events are frozen, the browser continues to operate normally and still generates events internally, but your application will not be notified of them. The events are not queued while frozen; they are discarded. For developers familiar with COM controls, this mirrors the behavior of `IOleControl::FreezeEvents`. When you unfreeze, only new events that occur after unfreezing will be delivered.

Event freezing can be useful when:

- You are running scripted or programmatic operations that would otherwise flood your event handler, such as multiple page redirects or rapid **WebViewAsyncExecute** calls.
- You are making a series of user interface changes and want to avoid reentrant updates. For example, temporarily stopping focus and/or size change events while moving or resizing controls.
- You are showing a modal dialog or performing work where event reentrancy could complicate state management.
- You need to suppress event-driven logic during controlled navigation, such as "silently" reloading a document or navigating to a new location.

Freezing does not alter browser state, navigation, or timers; it only affects your application's callback. To resume normal notifications, call this function with the ***bFreezeEvents*** parameter as zero. Freezing events will not affect calls to **WebViewWaitForEvent**. Because freezing events does not prevent browser events from occurring, they are still processed internally and you can wait for a specific event to complete.

If you discover that subsequent notifications within an event handler would interfere with your application's workflow, it is safe to freeze events from within the handler. After completing the

operation, unfreeze events (for example, by posting an application-defined message or using a timer) so events resume normally. You should avoid calling **WebViewDisableEvents** from within an event handler, use freezing instead.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewDisableEvents, WebViewEnableEvents, WebViewEventProc, WebViewWaitForEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetAllOptions Function

```
BOOL WINAPI WebViewGetAllOptions(
    HWND hWndHost,
    LPDWORD lpdwOptions
);
```

The **WebViewGetAllOptions** function returns the current option flags for the instance.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpdwOptions

A pointer to an unsigned integer which will contain the browser options for the current session. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function returns the options for the current session. For more information about the options available, review the WebView Options documentation. Always check the return value from this function; the option bitflags are only valid if this function returns a non-zero (TRUE) value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewGetOption, WebViewSetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetBackgroundColor Function

```
BOOL WINAPI WebViewGetBackgroundColor(
   HWND hWndHost,
   COLORREF *lprbgBackground
);
```

The **WebViewGetBackgroundColor** function returns the background color used behind the page content.

Parameters

hWndHost

Handle to the window attached to the browser session.

lprbgBackground

A pointer to an integer which will contain the current background color as an RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The background color specifies the fill used by the browser window when there is no content to display or the document itself does not define a background. This value only applies to the WebView host surface and does not affect page transparency. The alpha channel is ignored, and the color is treated as fully opaque. By default, the background color for new browser instances will be white.

A **COLORREF** is a 32-bit value used by the Windows API to represent an RGB color. The low-order byte specifies the red component, the next byte the green component, and the third byte the blue component, in the format **0x00BBGGRR**. The high-order byte is unused and should be zero. Note that **COLORREF** is specific to the Windows API and is not the same as the OLE color system (**OLE_COLOR**). OLE uses a slightly different encoding that allows system color values to be specified, which are resolved at runtime. Developers should be careful not to confuse the two.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewSetBackgroundColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetBrowserVersion Function

```
INT WINAPI WebViewGetBrowserVersion(
    LPDWORD lpdwVersionMS,
    LPDWORD lpdwVersionLS,
    LPTSTR lpszVersion,
    INT nMaxLength
);
```

The **WebViewGetBrowserVersion** function returns the installed browser runtime version information.

Parameters

lpdwVersionMS

A pointer to an unsigned integer which contains the major and minor version numbers. This parameter may be NULL.

lpdwVersionLS

A pointer to an unsigned integer which contains the build and release version numbers. This parameter may be NULL.

lpszVersion

A null-terminated string buffer which will contain the version as a printable string. This parameter may be NULL, in which case the version string is not copied but the function will still return the length of the version string if successful. If this parameter is NULL, the ***nMaxLength*** parameter should have a value of zero.

nMaxLength

An integer which specifies the maximum number of characters which may be copied into the string buffer, including the terminating null. If the ***lpszVersion*** parameter is NULL, this parameter should be zero.

Return Value

If the function succeeds, the return value is the length of the version string, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetBrowserVersion** function returns the version information for the installed Microsoft Edge WebView2 Runtime. This is the runtime that hosts the embedded browser control, and it is separate from the version of the Edge browser that may be installed on the system. The function can provide the version number as two 32-bit values (major/minor and build/revision), and optionally as a formatted string for display purposes. This function is useful for diagnostics, logging, or when an application requires a specific minimum runtime version to support certain features. If the WebView2 Runtime is not installed or cannot be located, the function will fail and return zero.

The version values returned by ***lpdwVersionMS*** and ***lpdwVersionLS*** can be used with the `HIWORD` and `LOWORD` macros to extract the individual components of the version number (for example, major, minor, build, and revision). The printable version string will match the same four-part format commonly shown in Edge and WebView2 documentation (e.g., "`126.0.2592.113`").

The result is independent of whether a WebView window has already been created or attached. It only reflects the version of the installed runtime and does not indicate whether the runtime is currently in use. For cases where you only need to check if a runtime is available, use the

WebViewRuntimeAvailable function instead.

Example

```
DWORD dwVerMS = 0, dwVerLS = 0;
TCHAR szVersion[64] = {0};

INT nLength = WebViewGetBrowserVersion(&dwVerMS, &dwVerLS, szVersion,
_countof(szVersion));
if (nLength > 0)
{
    WORD wMajor = HIWORD(dwVerMS);
    WORD wMinor = LOWORD(dwVerMS);
    WORD wBuild = HIWORD(dwVerLS);
    WORD wRevision = LOWORD(dwVerLS);

    _tprintf(_T("WebView2 Runtime Version: %s\n"), szVersion);
    _tprintf(_T("Major=%hu Minor=%hu Build=%hu Revision=%hu\n"), wMajor, wMinor,
wBuild, wRevision);
}
else
{
    _tprintf(_T("Unable to determine WebView2 Runtime version (Error 0x%lx)\n"),
WebViewGetLastError());
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewRuntimeAvailable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetChildWindow Function

```
HWND WINAPI WebViewGetChildWindow(
     HWND hWndHost
);
```

The **WebViewGetChildWindow** function returns the handle of the internal browser child window.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is a window handle. If the function fails, the return value is NULL. To get extended error information, call **WebViewGetLastError**.

Remarks

The value returned by this function is the handle to the internal child window that hosts the WebView control. Applications should treat this handle as an implementation detail and avoid sending messages or altering window styles directly. The control manages this child window and relies on its state for proper operation. Interfering with the child window (for example, by subclassing it, changing its parent, or destroying it) can cause undefined behavior and application instability.

This function is primarily provided to support advanced scenarios where access to the native window handle is unavoidable, such as integrating with APIs that require a child window handle. In most cases, developers should use the higher-level functions in the WebView API instead of interacting with the child window directly.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewFindWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetContentType Function

```
INT WINAPI WebViewGetContentType(
   HWND hWndHost
   LPTSTR lpszContentType,
   INT nMaxLength
);
```

The **WebViewGetContentType** function returns the content type for the current page.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszContentType

A string buffer which will contain the MIME content type of the resource returned by the web server. The buffer must be large enough to contain the entire content type string including the terminating null character. If this parameter is NULL, the content type will not be copied but the function will return the length of the `Content-Type` header value.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer, including the terminating null. If the *lpszContentType* parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the length of the `Content-Type` header value. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetContentType** function returns the MIME type reported by the server in the `Content-Type` response header for the most recent navigation. This value reflects the server's indication of what kind of content is being delivered, such as "`text/html`", "`application/pdf`", or "`image/png`". The returned string is preserved until the next navigation begins, even if the document is cleared or headers are reset.

It is important to note that not all servers will provide a `Content-Type` header. Some intentionally omit this information in combination with a `X-Content-Type-Options: nosniff` header as a security measure to prevent the client from "sniffing" or inferring the type. In such cases, **WebViewGetContentType** will return an empty string. Applications should be prepared for this scenario and not assume a value will always be available.

This function is a convenience wrapper around the underlying response header and is useful for distinguishing whether the WebView is rendering standard HTML or other document types. If more detailed information is needed about the response, developers can enumerate the response headers directly using **WebViewGetFirstHeader** and **WebViewGetNextHeader**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewClearHeaders, WebViewGetFirstHeader, WebViewGetHeader, WebViewGetNextHeader, WebViewSetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetCurrentUrl Function

```
BOOL WINAPI WebViewGetCurrentUrl(
    HWND hWndHost,
    LPTSTR lpszUrl,
    INT nMaxLength
);
```

The **WebViewGetCurrentUrl** function returns the URL for the current document.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszUrl

A string buffer which will contain the current URL when the function returns. If this parameter is NULL, the function will not copy anything into the buffer but will return the length of the URL, not including the terminating null character. If a buffer is provided, it must be large enough to contain the entire URL.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null. If the ***lpszUrl*** parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the length of the current URL, not including the terminating null. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetCurrentUrl** function retrieves the URL associated with the current document loaded in the browser instance. The URL value is available as soon as navigation begins to a new location, so it can be queried during or after the navigation process. This allows applications to obtain the destination address immediately, even before the page has fully loaded or rendered its content.

Note that the current URL may differ from the value originally passed to **WebViewNavigateUrl**. The browser engine can normalize the address, such as adjusting letter case, expanding relative paths, or adding default components like a trailing slash. In addition, the current URL can change multiple times during the course of a navigation if the server issues one or more HTTP redirects before the final document is retrieved. For this reason, the URL returned by this function should be treated as the authoritative location of the content currently being displayed.

This function differs from examining the document itself (for example, by executing a script to query `document.URL`) because it reflects the navigation state maintained internally by the control. Applications should use this function to reliably track the current location of the browser instance.

If the **WebViewSetDocumentText** function was called to change the current document, the URL returned by this function will typically be `"data:text/html"` which indicates it is a local data source.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewNavigateUrl, WebViewNormalizeUrl, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetDocumentText Function

```
INT WINAPI WebViewGetDocumentText(
    HWND hWndHost
    LPTSTR lpszText,
    INT nMaxLength
);
```

The **WebViewGetDocumentText** function returns the current document text.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszText

A string buffer which will contain the current document text when the function returns. If this parameter is NULL, the function will not copy anything into the buffer but will return the length of the text, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null. If the ***lpszText*** parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the length of the document text in characters, not including the terminating null. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetDocumentText** function retrieves the HTML markup for the current document loaded in the browser. This includes the complete contents of the `<html>` element, such as the `<head>` and `<body>` sections, rather than just the visible text.

The text returned will reflect the current state of the document at the time the function is called. If the page has been dynamically modified by scripts after it was initially loaded, those changes will be included. The value may be large, particularly for complex pages, and applications should allocate a buffer of sufficient size by first calling the function with a NULL pointer to determine the required length.

This function should only be called after navigation has completed. If called while a page is still loading, the results may be incomplete or empty. When the content is provided by a local file or string, the returned HTML will be the original content that was passed to the control. The returned text is raw HTML and may contain scripts, style definitions, and markup that are not directly suitable for display in a plain-text context.

If a blank page is currently being displayed, such as when navigating to `about:blank` or by setting the current document text to an empty string, this function can still return a value. Internally, the browser creates a blank page using the markup `"<html><head></head><body></body></html>"` and that is what this function will return.

There are two versions of this function, one that returns 8-bit text (**WebViewGetDocumentTextA**) and one that returns 16-bit Unicode text (**WebViewGetDocumentTextW**). The correct version of the function is called based on your project settings and whether the `UNICODE` macro is defined. If you call

the ANSI version of this function, the text that is returned will always be UTF-8 encoded, regardless of the current locale. If the document contains anything other than ASCII characters, it is recommended that you use the Unicode version of this function.

The **WebViewGetFilteredText** function can be used to obtain a filtered version of the text with the HTML markup removed.

Example

```
INT nLength = WebViewGetDocumentText(hWndHost, NULL, 0);
if (nLength > 0)
{
    LPTSTR pszBuffer = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (!pszBuffer)
    {
        // Virtual memory exhausted
        return;
    }

    nLength = WebViewGetDocumentText(hWndHost, pszBuffer, nLength + 1);
    if (nLength > 0)
    {
         // Write the document HTML to standard output
        _putts(pszBuffer);
        _puttc(_T('\n'), stdout);
    }

    free(pszBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetDocumentTitle, WebViewGetFilteredText, WebViewNavigateUrl, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetDocumentTitle Function

```
INT WINAPI WebViewGetDocumentTitle(
    HWND hWndHost,
    LPTSTR lpszTitle,
    INT nMaxLength,
);
```

The **WebViewGetDocumentTitle** function returns the document title.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszTitle

A string buffer which will contain the current document title when the function returns. If this parameter is NULL, the function will not copy anything into the buffer but will return the length of the title, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null. If the ***lpszTitle*** parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the length of the document title in characters, not including the terminating null. If the document does not have a title or the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetDocumentTitle** function returns the current title of the document loaded in the browser. This corresponds to the contents of the `<title>` element in the document's `<head>` section, if one is defined.

The title is not fixed at the time the page is first loaded. A script running in the document can modify the `<title>` element dynamically, and some web applications use this technique to reflect state changes (for example, to indicate new messages or status updates). Because of this, the value returned by this function may change over the lifetime of the page. If the document does not explicitly define a title, the function may return an empty string. Applications that display document titles should be prepared to handle this case.

If the browser window was created using **WebViewCreateWindow** or **WebViewCreateWindowEx** without a specific title, the application title will be automatically changed to reflect the current document title. This does not happen if the browser window is attached to an existing window or if an application title was specified when the window was created. If you wish to manually update a top-level window title with the current document title, call this function in response to a `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event and then call the **SetWindowText** function to change the text in the window's title bar.

If the current document does not have a title, this function will return zero and the last error code will be set to `NO_ERROR`. If the ***lpszTitle*** parameter is not NULL, it will be set to an empty string when the function returns. This is not uncommon if the **WebViewSetDocumentText** function is called to change the current document and the text does not contain HTML markup or does not include a `<title>` tag.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetDocumentText, WebViewNavigateUrl, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetDownloadInfo Function

```
BOOL WINAPI WebViewGetDownloadInfo(
   HWND hWndHost
   DWORD dwFileId,
   WEBVIEW_DOWNLOAD_INFO *lpInfo
);
```

The **WebViewGetDownloadInfo** function returns information about the status of the specified download.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwFileId

An unsigned integer value which uniquely identifies the download. This value cannot be zero.

lpInfo

A pointer to a `WEBVIEW_DOWNLOAD_INFO` structure which will contain information about the specified download when the function returns. The *dwSize* member of this structure must be initialized to the size of the structure prior to being passed to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function returns information about a specific download and is typically called in the context of a download event handler. The *dwFileId* parameter identifies the download to be queried. This identifier can be obtained either through download-related event notifications, where it is provided as the *lParam* value, or by calling the **WebViewEnumDownloads** function to enumerate active and completed downloads.

The file identifier should be treated as an opaque value. While it is guaranteed to be unique for the lifetime of the client session, the underlying implementation simply assigns sequential values. Applications must not depend on this behavior, as the specific numbering scheme may change in future versions. Because the identifiers are only unique within a single session, they cannot be used after the browser instance has been detached or destroyed. If you need to track downloads across sessions, you should use the file name or URL rather than the internal identifier.

Information is returned in the `WEBVIEW_DOWNLOAD_INFO` structure, which includes the download status, number of bytes copied, total size, elapsed time, file name, and the source URL. If the specified identifier does not correspond to a valid download, the function will fail.

If you are only interested in the name of the local file used for the specified download, you can use the **WebViewGetDownloadPath** function.

Example

```
// Get the total number of downloads
INT nTotal = WebViewEnumDownloads(hWndHost, 0, NULL, 0);

if (nTotal <= 0)
```

```
{
    _tprintf(_T("No downloads found.\n"));
    return;
}

// Allocate a buffer for the identifiers
DWORD *pIds = (DWORD*)malloc(nTotal * sizeof(DWORD));
if (!pIds)
    return;

INT nCount = WebViewEnumDownloads(hWndHost, 0, pIds, nTotal);

// Enumerate each download and print basic info
for (INT nIndex = 0; nIndex < nCount; nIndex++)
{
    WEBVIEW_DOWNLOAD_INFO fileInfo = { sizeof(fileInfo) };

    if (WebViewGetDownloadInfo(hWndHost, pIds[nIndex], &fileInfo))
    {
        _tprintf(_T("Download %u: %s (%llu/%llu bytes)\n"),
                 fileInfo.dwFileId,
                 fileInfo.szFileName,
                 fileInfo.uiBytesCopied.QuadPart,
                 fileInfo.uiBytesTotal.QuadPart);
    }
}

free(pIds);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewCancelDownload, WebViewEnumDownloads, WebViewGetDownloadPath, WebViewSetDownloadPath, WEBVIEW_DOWNLOAD_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetDownloadPath Function

```
INT WINAPI WebViewGetDownloadPath(
    HWND hWndHost,
    DWORD dwFileId,
    LPTSTR lpszFilePath,
    INT nMaxLength
);
```

The **WebViewGetDownloadPath** function returns the complete path of the local file for the specified download.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwFileId

An unsigned integer value which uniquely identifies the download. This value cannot be zero.

lpszFilePath

A null-terminated string buffer which will contain the full path to the local file. If this parameter is NULL, the file path is not copied but the function will return the length of the file path, not including the terminating null character. It is recommended that this buffer be large enough to store `MAX_PATH` (260) characters.

nMaxLength

An integer value which specifies the maximum number of characters which may be copied into the string buffer, including the terminating null. If the ***lpszFilePath*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the length of the local file name, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function returns the complete path to the file that will be created or overwritten during the download. The file path returned by this function is always an absolute path which ensures that the application does not need to resolve relative paths or expand environment variables. The path returned may refer to a file that has not yet been created or fully written if the download is still in progress. Applications should check the download status with **WebViewGetDownloadInfo** before assuming the file is complete and ready to use.

The ***dwFileId*** parameter uniquely identifies the download within the current client session. This identifier can be obtained from download-related event notifications or by calling **WebViewEnumDownloads**. As with other functions that use file identifiers, applications should treat this value as opaque and only valid for the lifetime of the session.

If the provided string buffer is too small, the function will return the required length of the file path, not including the terminating null character. The caller can use this information to allocate a sufficiently large buffer and call the function again. There are both ANSI and Unicode versions of this function and the correct version is called based on how the project is configured. If the **WebViewGetDownloadPathA** function is called, the file path will always be returned as a UTF-8

encoded string regardless of the current locale.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewCancelDownload, WebViewEnumDownloads, WebViewGetDownloadInfo, WebViewSetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetErrorString Function

```
INT WINAPI WebViewGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **WebViewGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **WebViewGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetLastError, WebViewSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetFilteredText Function

```
INT WINAPI WebViewGetFilteredText(
   HWND hWndHost,
   LPTSTR lpszFilteredText,
   INT nMaxLength
);
```

The **WebViewGetFilteredText** function filters HTML formatting from the current document and returns plain text in a string buffer.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszFilteredText

A null-terminated string buffer which will contain the filtered text. If this parameter is NULL and the *nMaxLength* parameter is zero, the function will return the number of characters required to store the filtered text, not including the terminating null character.

nMaxLength

The maximum number of characters which can be copied into the *lpszFilteredText* string buffer. If this value is zero, no characters will be copied to the buffer and the function will return the length of the filtered text, not including the terminating null character. This can be used to determine how much memory should be allocated for the string.

Return Value

If the function succeeds, it will return the number of characters copied into the string buffer, not including the terminating null character. If the *lpszFilteredText* parameter is NULL and *nMaxLength* is zero, the function will return the number of characters in the filtered text without copying any characters. If the function fails, or there is no text in the input buffer, the function will return zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetFilteredText** function filters the HTML markup from the document text and returns output suitable for display in text-only environments. HTML tags are removed and entities are converted to their canonical Unicode equivalent. When filtering markup from the text, the function will:

- Remove all leading and trailing whitespace from the text, including newlines.
- Remove script, style, and other non-visible or unsafe elements.
- Convert common tag structures, such as lists and headers, into readable plain text.
- Extract and filter hyperlinks and image alt text, ignoring internal links to other sections.
- Remove any embedded credentials in hyperlinks.
- Decode common HTML entities such as `&trade` and `&copy`, as well as numeric entities like `&#x20.`

The function will remove extraneous whitespace within the text, but will attempt to maintain the structure of ordered and unordered lists. It does not parse the style information or Javascript that may be included in the HTML content. If a paragraph uses a style that would normally prevent it from being rendered in a browser (e.g.: the visibility attribute set to "hidden"), it is ignored and the text will be included in the output. If there are sections which contain preformatted text, such as text inside

`<pre>...</pre>` or `<code>...</code>` tags, individual line and paragraph breaks will be preserved.

This function will not remove Unicode characters from the input text. If the HTML markup includes any entities which are represented as Unicode, such as `½` which represents "½", they will be returned in the output string buffer as Unicode characters. If the ANSI version of this function is called, Unicode will always be returned as UTF-8 encoded characters.

Example

```
INT nLength = WebViewGetFilteredText(hWndHost, NULL, 0);
if (nLength > 0)
{
    LPTSTR pszFiltered = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (!pszFiltered )
    {
        // Virtual memory exhausted
        return;
    }

    nLength = WebViewGetFilteredText(hWndHost, pszFiltered, nLength + 1);
    if (nLength > 0)
    {
         // Write the plain text to standard output
        _putts(pszFiltered);
        _puttc(_T('\n'), stdout);
    }

    free(pszFiltered);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetDocumentText, WebViewPrintDocument, WebViewSaveDocument, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetFirstHeader Function

```
BOOL WINAPI WebViewGetFirstHeader(
    HWND hWndHost,
    LPTSTR lpszHeaderName,
    INT nMaxNameLen,
    LPTSTR lpszHeaderValue,
    INT nMaxValueLen
);
```

The **WebViewGetFirstHeader** function begins enumerating response headers and return the first name/value pair.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszHeaderName

A string buffer which will contain the name of the first header returned by the server. If this parameter is NULL it will be ignored.

nMaxNameLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderName* string buffer, including the terminating null. If the *lpszHeaderName* parameter is NULL this value should be zero.

lpszHeaderValue

A string buffer which will contain the value of the first header returned by the server. If this parameter is NULL, it will be ignored.

nMaxValueLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderValue* string buffer, including the terminating null. If the *lpszHeaderValue* parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The response headers exposed by this function correspond to the most recent completed navigation for the specified browser instance. Callers typically use this function to start an enumeration and then call **WebViewGetNextHeader** repeatedly until it returns FALSE to indicate there are no more headers to enumerate.

Header names are case-insensitive by definition. This function will usually return header names as lower-case values, but applications should never depend on the casing and should compare names case-insensitively. The header names and values are returned exactly as enumerated internally by the WebView2 control.

Some response headers can appear multiple times in a message. For list-valued headers (those defined by the HTTP specifications to be comma-joinable), the implementation may return a single entry with a comma-separated value (e.g., `Accept-Language`, `Cache-Control`, etc.). Other headers

must not be combined; for example, each **Set-Cookie** header is returned as a separate name/value pair and will be seen multiple times during enumeration, each with a single cookie value. Code that depends on specific headers should be prepared for either a single combined value or multiple discrete occurrences of the same header name, depending on the header's semantics.

Enumeration reflects a snapshot of the headers for that response. Beginning a new navigation or calling **WebViewClearHeaders** invalidates the current enumeration. Only one header enumeration can be active per browser instance at a time. The enumeration state is not thread-safe, and any access to the response headers should be synchronized across different threads.

If the provided buffers are too small, the header names and values will be truncated to the available space specified by the maximum length parameters. In most cases, a header name is not going to exceed 256 characters and a header value will normally not exceed 5120 characters. Most header values are relatively short, but some headers such as **Set-Cookie** can return much longer values.

This function cannot be called while the browser is in the process of navigating to a new location. It will also fail if the current document is a local file or if the document text was set dynamically using **WebViewSetDocumentText**. There are no response headers available in a local context.

Example

```
TCHAR szName[WEBVIEW_MAX_HEADER_NAME];
TCHAR szValue[WEBVIEW_MAX_HEADER_VALUE];

if (WebViewGetFirstHeader(hWndHost,
                          szName, _countof(szName),
                          szValue, _countof(szValue)))
{
    do
    {
        _tprintf(_T("%s: %s\n"), szName, szValue);
    }
    while (WebViewGetNextHeader(hWndHost,
                                szName, _countof(szName),
                                szValue, _countof(szValue)));
}
else
{
    DWORD dwError = WebViewGetLastError();
    _tprintf(_T("No headers available (error 0x%lx)\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewClearHeaders, WebViewGetHeader, WebViewGetNextHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetFocus Function

```
BOOL WINAPI WebViewGetFocus(
    HWND hWndHost,
    BOOL *lpbHasFocus
);
```

The **WebViewGetFocus** function returns whether the browser has focus.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpbHasFocus

A pointer to a BOOL which will contain TRUE (non-zero) if the browser window has focus and FALSE (zero) if it does not have focus. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetFocus** function queries whether the browser control currently has input focus. This is useful for determining if keyboard input will be directed to the content area of the browser or handled by the host window. The focus state can change dynamically as the user interacts with the application, and applications that track focus should also monitor the `WEBVIEW_EVENT_FOCUS_CHANGED` event.

This function only reports the focus state of the embedded browser, not of the host window itself. For example, a host window may be the active foreground window, but if another control within the host has focus, the browser will not be considered focused. To explicitly give focus to the browser, call **WebViewSetFocus**.

Example

```
BOOL bHasFocus = FALSE;

if (WebViewGetFocus(hWndHost, &bHasFocus))
{
    if (!bHasFocus)
    {
        // Give the browser control focus if it does not have it
        WebViewSetFocus(hWndHost);
    }
}
else
{
    // Handle error
    DWORD dwError = WebViewGetLastError();
    _tprintf(_T("WebViewGetFocus failed, error=0x%lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewRedrawWindow, WebViewIsVisibile, WebViewSetFocus, WebViewSetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetHeader Function

```
INT WINAPI WebViewGetHeader(
   HWND hWndHost,
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue,
   INT nMaxLength
);
```

The **WebViewGetHeader** function returns the value of a specific response header by name.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszHeaderName

A null-terminated string which specifies the response header name. This value is not case-sensitive but must match the complete header name. This parameter cannot be NULL.

lpszHeaderValue

A string buffer which will contain the value of the specified header. If this parameter is NULL then nothing will be copied but the function will return the length of the header value, not including the terminating null.

nMaxLength

The maximum number of characters which may be copied into the string buffer, including the terminating null. If the ***lpszHeaderValue*** parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is the length of the header value, not including the terminating null. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetHeader** function retrieves the value of a specific HTTP response header returned by the server. Header names are matched in a case-insensitive manner, with leading and trailing whitespace automatically trimmed before comparison. If multiple values are present for the specified header name, the behavior depends on the type of header:

- For headers that must not be combined (such as `Set-Cookie` and `Proxy-Authenticate`), only the first value encountered is returned.
- For all other headers, multiple values are joined together and returned as a single comma-separated string, as defined by the HTTP specification.

For headers such as `Set-Cookie`, which may legitimately occur multiple times with distinct values, **WebViewGetHeader** will only return the first occurrence. If your application needs to process each value individually, you should use **WebViewGetFirstHeader** and **WebViewGetNextHeader** to enumerate all response headers rather than relying on this function.

If the ***lpszHeaderValue*** parameter is NULL, the function returns the number of characters required to store the complete header value, excluding the terminating null. This enables applications to dynamically allocate a buffer of the correct size before retrieving the actual value.

Response headers are only available after navigation has completed. Calling this function while a navigation is still in progress, or when displaying local file or data content, will fail.

The maximum recommended length for a header name is **WEBVIEW_MAX_HEADER_NAME** (256 characters) and for a header value is **WEBVIEW_MAX_HEADER_VALUE** (5120 characters). These limits are intended as practical guidelines, not absolute restrictions.

Example

```
 INT nLength = WebViewGetHeader(hWndWebView, _T("Content-Type"), NULL, 0);
if (nLength <= 0)
{
    _tprintf(TEXT("Unable to get Content-Type header (error 0x%lx)\n"),
WebViewGetLastError());
    return;
}

LPTSTR lpszValue = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
if (!lpszValue)
{
    _tprintf(_T("Out of memory\n"));
    return;
}

if (WebViewGetHeader(hWndWebView, _T("Content-Type"), lpszValue, nLength + 1) > 0)
{
    _tprintf(_T("Content-Type: %s\n"), lpszValue);
}
else
{
    _tprintf(_T("Failed to retrieve Content-Type (error 0x%lx)\n"),
WebViewGetLastError());
}

free(lpszValue);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewClearHeaders, WebViewGetFirstHeader, WebViewGetNextHeader, WebViewSetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetHeaderCount Function

```
BOOL WINAPI WebViewGetHeaderCount(
    HWND hWndHost
    DWORD dwHeaderType,
    LONG *lpnCount
);
```

The **WebViewGetHeaderCount** function returns the number of headers defined for the specified header type.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwHeaderType

An unsigned integer which specifies the type of header being checked. It may be one of the following values.

Name	Description
WEBVIEW_HEADERS_REQUEST	The number of custom headers submitted with a request.
WEBVIEW_HEADERS_RESPONSE	The number of headers returned by the server.

lpnCount

A pointer to an integer which will contain the number of headers when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function can be used to determine the number of custom request headers which have been set, or the number of headers returned by the server in response to a navigation. To enumerate the available response headers, use the **WebViewGetFirstHeader** and **WebViewGetNextHeader** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearHeaders, WebViewGetFirstHeader, WebViewGetNextHeader, WebViewSetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetHttpStatus Function

```
INT WINAPI WebViewGetHttpStatus(
    HWND hWndHost
);
```

The **WebViewGetHttpStatus** function returns the HTTP status code from the last completed navigation.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is a non-zero status code. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function returns the HTTP status code from the server response associated with the last completed navigation. These codes are defined by the HTTP standard and indicate the outcome of the request:

- **100-199**: Informational responses indicating the request is in progress.
- **200-299**: Successful responses, with 200 (OK) being the most common.
- **300-399**: Redirects, such as 301 (Moved Permanently) or 302 (Found).
- **400-499**: Client errors, such as 403 (Forbidden) or 404 (Not Found).
- **500-599**: Server errors, such as 500 (Internal Server Error) or 503 (Service Unavailable).

It is important to note that this value is only meaningful once navigation has completed. During a series of redirects, the status code may change multiple times; the value returned will be from the final response that completed navigation. A return value of **0** indicates the HTTP status code is unavailable. It may also return zero when the browser has navigated to a local file or when HTML content has been supplied directly using the **WebViewSetDocumentText** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetNavigationStatus, WebViewGetSecurityLevel, WebViewNavigateUrl, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetInterface Function

```
BOOL WINAPI WebViewGetInterface(
   HWND hWndHost,
   UINT_PTR lpInterface
);
```

The **WebViewGetInterface** function returns a raw COM interface pointer to the WebView2 browser control.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpInterface

A pointer to the COM interface for the browser control.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetInterface** function returns the underlying **ICoreWebView2** COM interface pointer that is associated with the specified browser instance. This function is provided for advanced scenarios where direct access to the WebView2 SDK is required. Most applications will not need to call this function and should instead use the higher-level functions in this API.

If the function succeeds, the interface pointer is returned through the *lpInterface* parameter. The pointer is not reference-counted on return. If your code intends to use the **ICoreWebView2** interface directly, you *must* increment the reference count by calling `AddRef` before wrapping it in a smart pointer such as `ComPtr<ICoreWebView2>`. Failure to do so can result in undefined behavior if the browser instance is later detached or destroyed.

Because this function exposes the raw COM interface, any code that uses it must include the Microsoft.Web.WebView2 SDK headers and link against the appropriate libraries. This introduces a dependency on the WebView2 SDK that is not otherwise required when using this library.

In most cases, this function is only used to determine whether a browser instance has been attached to a host window, treating the returned `UINT_PTR` value as opaque. Developers who choose to interact with the **ICoreWebView2** interface directly should be familiar with COM programming practices, including reference counting and apartment threading rules.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetLastError Function

```
DWORD WINAPI WebViewGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **WebViewGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **WebViewSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE or NULL. Those functions which call **WebViewSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetErrorString, WebViewSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetMessage Function

```
BOOL WINAPI WebViewGetMessage(
    HWND hWndHost.
    LPTSTR lpszMessage,
    INT nMaxLength
);
```

The **WebViewGetMessage** function retrieves the next pending message sent from the current document.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszMessage

A null-terminated string buffer which will contain the message sent from the current document. If this parameter is NULL, the message will not be copied and the function will return the length of the message, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the ***lpszMessage*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters in the message, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetMessage** function retrieves the most recent message sent from script code running in the browser. This is typically used to enable interaction between the web content displayed in the browser and the host application. For example, a page might use notify the application when a user completes a form, clicks a button, or when some script logic produces a result that the native application needs to act upon. This provides a simple and secure channel for exchanging data between the web page and the host, without requiring the page to reload or navigate to a new URL.

Messages are sent from JavaScript using the following method:

```
window.chrome.webview.postMessage(value);
```

When a script in the browser sends a message, the `WEBVIEW_EVENT_MESSAGE_RECEIVED` event will fire and your event handler should call this function to retrieve the value posted by the script. The `value` parameter can be any JSON-serializable object or a simple string. The format of the message depends on the type of value that was posted:

- If the message was a structured JavaScript object or array, it will be returned as a JSON object string using a format like `{"key":"value"}`.
- If the message was a string, it will be returned as a JSON-formatted string with surrounding quotes and any special characters escaped, such as `"Hello\nWorld"`.

Applications that need to process these messages should parse the result as JSON to properly distinguish between plain strings and structured data.

Important: Message passing is disabled by default. To enable this functionality, the browser must be created or attached with the `WEBVIEW_OPTION_ENABLE_MESSAGES` flag. If this option is not specified, no messages will be received and this function will not return any content.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewGetDocumentText, WebViewGetResult, WebViewPostMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetNavigationStatus Function

```
BOOL WINAPI WebViewGetNavigationStatus(
    HWND hWndHost
    LPDWORD lpdwFlags,
    LPDWORD lpdwError
);
```

The **WebViewGetNavigationStatus** function returns the current navigation state.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpdwFlags

A pointer to an unsigned integer which may contain one of more status bitflags. This parameter may be NULL if the status flags are not required.

Name	Description
WEBVIEW_NAVIGATION_REDIRECTED	The navigation was the result of an HTTP redirect.
WEBVIEW_NAVIGATION_USER_INITIATED	Navigation began from a user gesture rather than being initiated by a script.
WEBVIEW_NAVIGATION_LOCAL_DATA	Content was served using local data rather than a full network fetch.

lpdwError

A pointer to a unsigned integer which will contain one of the following error codes. This parameter may be NULL if the error code is not required.

Name	Description
WEBVIEW_ERROR_NONE	No error.
WEBVIEW_ERROR_CERTIFICATE_COMMON_NAME	Certificate common name mismatch.
WEBVIEW_ERROR_CERTIFICATE_EXPIRED	Certificate is expired.
WEBVIEW_ERROR_INVALID_CLIENT_CERTIFICATE	Invalid or untrusted client certificate.
WEBVIEW_ERROR_CERTIFICATE_REVOKED	Certificate has been revoked.
WEBVIEW_ERROR_INVALID_CERTIFICATE	Certificate is invalid for another reason.
WEBVIEW_ERROR_SERVER_UNREACHABLE	Cannot reach the server/host.
WEBVIEW_ERROR_TIMEOUT	Operation timed out.
WEBVIEW_ERROR_INVALID_SERVER_RESPONSE	Server returned an invalid or unexpected response.
WEBVIEW_ERROR_CONNECTION_ABORTED	Connection aborted by the

	server or network.
WEBVIEW_ERROR_CONNECTION_RESET	Connection was reset.
WEBVIEW_ERROR_DISCONNECTED	Disconnected during the operation.
WEBVIEW_ERROR_CANNOT_CONNECT	Failed to establish a connection.
WEBVIEW_ERROR_HOST_NAME_NOT_RESOLVED	DNS resolution failed for the host.
WEBVIEW_ERROR_OPERATION_CANCELED	Operation was canceled by the user or host.
WEBVIEW_ERROR_REDIRECT_FAILED	Redirection failed.
WEBVIEW_ERROR_UNEXPECTED_ERROR	An unexpected error occurred.
WEBVIEW_ERROR_CREDENTIALS_REQUIRED	Authentication credentials are required.
WEBVIEW_ERROR_PROXY_AUTHENTICATION_REQUIRED	Proxy authentication is required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetNavigationStatus** function provides detailed information about the current state of navigation for the specified WebView instance. It can be used to determine if a navigation is still in progress, whether it completed successfully, or if it was canceled or failed due to an error condition. The status flags returned in ***lpdwFlags*** can be combined to indicate additional context about the navigation, such as whether it was initiated by the user, if the content was served from local data rather than fetched over the network, or if the navigation resulted from an HTTP redirect.

Developers may find this function most useful in the following situations:

- After handling a `WEBVIEW_EVENT_NAVIGATION_COMPLETED` event, to check if the navigation was successful.
- Before retrying or redirecting a failed request, to determine the specific cause of failure.
- When logging diagnostic information about connectivity or certificate errors.
- To distinguish between user-initiated cancellation, network timeouts, or security errors.
- To detect when content was provided from local data rather than fetched remotely.

The error code returned in ***lpdwError*** provides additional detail when a navigation fails or is blocked. These values cover a wide range of conditions, including invalid or expired certificates, connectivity issues, unreachable hosts, proxy authentication failures, and other network-related problems. If ***lpdwError*** is `WEBVIEW_ERROR_NONE`, the most recent navigation completed without error.

When navigation involves local files or HTML provided directly to the browser via **WebViewSetDocumentText**, the function will typically return `WEBVIEW_ERROR_NONE` and may set `WEBVIEW_NAVIGATION_LOCAL_DATA` in the status flags. This allows the application to detect that the content was not fetched over the network.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewCancelNavigation, WebViewGetHttpStatus, WebViewGetNavigationTime, WebViewNavigateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetNavigationTime Function

```
BOOL WINAPI WebViewGetNavigationTime(
    HWND hWndHost,
    LPDWORD lpdwElapsed,
    LPBOOL lpbNavigating
);
```

The **WebViewFunctionName** function returns the elapsed time for the last navigation.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpdwElapsed

A pointer to an unsigned integer which will contain the number of milliseconds elapsed since the current navigation has started.

lpbNavigating

A pointer to an integer which is updated when the function returns. It will contain a non-zero value if the browser is in the process of navigating to a new location. If the browser has completed navigation or the last navigation was canceled, this value will be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetNavigationTime** function measures how long the most recent navigation has taken, reported in milliseconds. If a navigation is currently in progress, the elapsed time represents the duration since the navigation began. If the navigation has already completed, the elapsed time represents the total duration of that navigation. If no navigation has occurred, the elapsed time will be zero.

This function can be used for diagnostics, performance monitoring, or implementing application-level timeouts. For example, an application may track whether a page is taking unusually long to load and decide to cancel the navigation, notify the user, or retry the request. Combined with **WebViewGetNavigationStatus**, it provides a way to determine both the state and timing of navigation operations.

Typical usage scenarios for this function include:

- Measuring page load times for performance profiling or telemetry.
- Implementing a custom navigation timeout by checking the elapsed time in a periodic timer.
- Providing user feedback, such as displaying how long a page has been loading.
- Logging navigation duration along with success or failure status for diagnostic purposes.

The *lpbNavigating* parameter indicates whether a navigation is currently in progress when the function is called. This allows developers to distinguish between a completed navigation (where the elapsed time is final) and an in-progress navigation (where the elapsed time is still increasing).

It is important to note that the reported time is based on the browser's internal navigation start and completion timestamps. It reflects the overall duration of the navigation process as observed by the

browser, not fine-grained network-level timings such as DNS lookup, TCP connection, or TLS handshake. For most application scenarios this high-level timing is sufficient to evaluate page load performance and detect unusual delays.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetNavigationStatus, WebViewNavigateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetNextHeader Function

```
BOOL WINAPI WebViewGetNextHeader(
   HWND hWndHost,
   LPTSTR lpszHeaderName,
   INT nMaxNameLen,
   LPTSTR lpszHeaderValue,
   INT nMaxValueLen
);
```

The **WebViewGetNextHeader** function returns the next name/value pair in a response header enumeration.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszHeaderName

A string buffer which will contain the name of the first header returned by the server. If this parameter is NULL it will be ignored.

nMaxNameLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderName* string buffer, including the terminating null. If the *lpszHeaderName* parameter is NULL this value should be zero.

lpszHeaderValue

A string buffer which will contain the value of the first header returned by the server. If this parameter is NULL, it will be ignored.

nMaxValueLen

An integer which specifies the maximum number of characters which can be copied into the *lpszHeaderValue* string buffer, including the terminating null. If the *lpszHeaderValue* parameter is NULL this value should be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

This function continues the enumeration that was started when the **WebViewGetFirstHeader** function was called. Repeatedly call **WebViewGetNextHeader** until it returns zero (FALSE) indicating there are no more header values to return. The **WebViewGetLastError** function can be used to distinguish between an actual error condition and the end of the enumeration. If the last error code is `ST_ERROR_END_OF_DATA` then all response headers have been enumerated.

If the provided buffers are too small, the header names and values will be truncated to the available space specified by the maximum length parameters. In most cases, a header name is not going to exceed 256 characters and a header value will normally not exceed 5120 characters. Most header values are relatively short, but some headers such as `Set-Cookie` can return much longer values.

This function cannot be called while the browser is in the process of navigating to a new location. It will also fail if the current document is a local file or if the document text was set dynamically using

WebViewSetDocumentText. There are no response headers available in a local context.

Example

```
TCHAR szName[WEBVIEW_MAX_HEADER_NAME];
TCHAR szValue[WEBVIEW_MAX_HEADER_VALUE];

if (WebViewGetFirstHeader(hWndHost,
                          szName, _countof(szName),
                          szValue, _countof(szValue)))
{
    do
    {
        _tprintf(_T("%s: %s\n"), szName, szValue);
    }
    while (WebViewGetNextHeader(hWndHost,
                                szName, _countof(szName),
                                szValue, _countof(szValue)));
}
else
{
    DWORD dwError = WebViewGetLastError();
    _tprintf(_T("No headers available (error 0x%lx)\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewClearHeaders, WebViewGetFirstHeader, WebViewGetHeader, WebViewGetHeaderCount

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetOption Function

```
BOOL WINAPI WebViewGetOption(
    HWND hWndHost,
    DWORD dwOption
    LPBOOL lpbEnabled
);
```

The **WebViewGetOption** function checks whether a specific option flag is enabled.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwOption

An unsigned integer which specifies the option to be checked. Refer to the WebView Options documentation for the list of available options. Only a single option may be specified; you cannot combine multiple option bitflags when calling this function.

lpbEnabled

A pointer to an integer which will be updated when the function returns. If the value is non-zero, the option has been enabled. If the value is zero, the option has not been enabled.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetOption** function is used to determine if a specific option has been enabled for the browser instance associated with the specified host window. Each option corresponds to an individual feature or behavior that can be enabled or disabled at creation time or later using **WebViewSetOption**.

This function only supports checking a single option value at a time. The *dwOption* parameter must specify exactly one option bitflag, such as `WEBVIEW_OPTION_ENABLE_SCRIPTING` or `WEBVIEW_OPTION_ENABLE_CONTEXT_MENU`. It is not valid to combine multiple bitflags with a bitwise OR. If you need to retrieve the complete set of enabled options, call **WebViewGetAllOptions**.

Note that not all options may be applicable after a window has been created. Some options only take effect at creation time and cannot be changed dynamically. For example, options related to profile isolation or privacy modes must be specified before the control is initialized.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewGetAllOptions, WebViewSetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetResult Function

```
BOOL WINAPI WebViewGetResult(
    HWND hWndHost,
    LPTSTR lpszResult,
    INT nMaxLength
);
```

The **WebViewGetResult** function returns the result from the most recent completed script execution.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszResult

A null-terminated string buffer which will contain the result from the script that was previously executed. If this parameter is NULL, the result will not be copied and the function will return the length of the string, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the ***lpszResult*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters in the result string, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetResult** function retrieves the result of the most recent JavaScript expression executed in the browser using either the **WebViewAsyncExecute** or **WebViewExecute** functions. The result is provided as a JSON-formatted string, which may be either a quoted string with escaped characters or a JSON object, depending on the value returned by the script. Applications that expect structured data should parse the returned string using a JSON parser.

Only the result from the most recently executed script is retained. Executing another script will overwrite the previous result, even if the value has not yet been retrieved. If your application needs to preserve earlier results, it should copy the returned string into its own buffer for later use.

When using **WebViewAsyncExecute**, the result is available once the `WEBVIEW_EVENT_SCRIPT_COMPLETED` event has been raised. Attempting to call **WebViewGetResult** before the event is delivered will return an empty string. When using **WebViewExecute**, the script is executed synchronously and the result is immediately available.

The maximum length of the result string is limited by the size of the buffer provided. If the ***lpszResult*** parameter is NULL, the function will return the required length of the result string in characters, not including the terminating null, which can be used to allocate a buffer of the appropriate size.

Example

```
// Execute a simple script that returns a string
INT nLength = WebViewExecute(hWndHost, _T("document.title"), NULL, 0);

if (nLength > 0)
```

```
{
    // Allocate a buffer large enough to hold the result
    LPTSTR lpszResult = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));
    if (lpszResult)
    {
        // Retrieve the result of the script execution
        if (WebViewGetResult(hWndHost, lpszResult, nLength + 1))
        {
            _tprintf(_T("Page title is: %s\n"), lpszResult);
        }
        free(lpszResult);
    }
}
else
{
    _tprintf(_T("Script execution failed\n"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAsyncExecute, WebViewExecute, WebViewGetMessage

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetSecurityInfo Function

```
BOOL WINAPI WebViewGetSecurityInfo(
   HWND hWndHost,
   WEBVIEW_SECURITY_INFO *lpSecurityInfo
);
```

The **WebViewGetSecurityInfo** function returns certificate and host security details for the last navigation.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpSecurityInfo

A pointer to a `WEBVIEW_SECURITY_INFO` structure which will contain information about the specified download when the function returns. The ***dwSize*** member of this structure must be initialized to the size of the structure prior to being passed to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetSecurityInfo** function retrieves details about the security state of the current page and populates a `WEBVIEW_SECURITY_INFO` structure. This includes the negotiated TLS protocol, cipher suite, certificate subject and issuer, and the validity period of the certificate. The function also reports an overall security level (***dwLevel***) and condition-specific flags (***dwFlags***) that identify issues such as expired certificates, mixed content, or weak cryptographic algorithms.

The information returned reflects the most recent navigation state. During navigation, these structure values may be provisional. For example, a page may initially appear secure but later be downgraded to a warning state if mixed content is detected. Applications should consider the values as advisory until a `WEBVIEW_EVENT_NAVIGATION_COMPLETED` notification confirms the final state.

The security level (***dwLevel***) is intended for user interface decisions, such as whether to display a lock icon. The flags (***dwFlags***) provide more granular context and may include multiple indicators at once. For instance, a site may be considered secure overall but still include a `WEBVIEW_SECURITY_FLAG_CERT_EXPIRED` flag if the certificate is past its validity period. Because the browser can revise its assessment as resources are loaded, applications that display security indicators should refresh their state after navigation completes. Do not assume that an initially secure status will remain unchanged for the life of the page. Conditions such as mixed content or blocked subresources may alter the security level after the main document is retrieved.

For plain HTTP or local content, the ***szProtocol*** and ***szCipher*** members will be empty, ***dwFlags*** will typically include `WEBVIEW_SECURITY_FLAG_UNENCRYPTED`, and the security level will be `WEBVIEW_SECURITY_LEVEL_INSECURE` or `WEBVIEW_SECURITY_LEVEL_NEUTRAL`. For HTTPS with a valid certificate, the protocol and cipher suite are reported, the subject and issuer strings identify the certificate, and the security level will be `WEBVIEW_SECURITY_LEVEL_SECURE`.

The subject and issuer members are provided for display or logging purposes only; certificate parsing and validation are handled internally by the browser engine. Likewise, the validity dates (***ftValidFrom*** and ***ftValidTo***) are exposed for simple checks but do not require applications to perform their own

certificate validation.

Before calling this function, the ***dwSize*** member of the structure must be initialized to the size of the structure. This ensures compatibility with future versions of the library if additional fields are introduced. If the size does not match, the function will fail or only initialize the subset of fields it recognizes.

Both ANSI (A) and Unicode (W) variants are available. When compiling with UNICODE defined, string members are UTF-16 encoded; otherwise, they are UTF-8 encoded. The **WebViewGetSecurityInfo** macro resolves to the appropriate version based on the project's character set.

If you are only interested in determining the current security level of the connection you can use the **WebViewGetSecurityLevel** function.

Example

```
// Map security level to text
static LPCTSTR SecLevelToText(DWORD level)
{
    switch (level)
    {
    case WEBVIEW_SECURITY_LEVEL_SECURE: return _T("SECURE");
    case WEBVIEW_SECURITY_LEVEL_WARNING: return _T("WARNING");
    case WEBVIEW_SECURITY_LEVEL_INSECURE: return _T("INSECURE");
    case WEBVIEW_SECURITY_LEVEL_NEUTRAL: return _T("NEUTRAL");
    default: return _T("UNKNOWN");
    }
}

// Print a FILETIME value as YYYY-MM-DD HH:MM:SS
static void PrintFileTime(LPCTSTR label, const FILETIME* pft)
{
    if (!pft || (pft->dwLowDateTime == 0 && pft->dwHighDateTime == 0))
    {
        _tprintf(_T("%s: (not available)\n"), label);
        return;
    }

    FILETIME lft = {0};
    SYSTEMTIME st = {0};

    if (FileTimeToLocalFileTime(pft, &lft) && FileTimeToSystemTime(&lft, &st))
    {
        _tprintf(_T("%s: %04u-%02u-%02u %02u:%02u:%02u\n"),
                 label,
                 (unsigned)st.wYear,
                 (unsigned)st.wMonth,
                 (unsigned)st.wDay,
                 (unsigned)st.wHour,
                 (unsigned)st.wMinute,
                 (unsigned)st.wSecond);
    }
    else
    {
        _tprintf(_T("%s: (conversion failed)\n"), label);
    }
}

// Build and print a readable flag list
```

```
static void PrintSecurityFlags(DWORD dwFlags)
{
    if (dwFlags == WEBVIEW_SECURITY_FLAG_NONE)
    {
        _tprintf(_T("Flags: NONE (0x%08lX)\n"), dwFlags);
        return;
    }

    _tprintf(_T("Flags: 0x%08lX"), dwFlags);

    if (dwFlags & WEBVIEW_SECURITY_FLAG_UNENCRYPTED) _tprintf(_T(" |
UNENCRYPTED"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_INSECURE) _tprintf(_T(" | INSECURE"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_CERT_WEAK) _tprintf(_T(" | CERT_WEAK"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_CERT_ERROR) _tprintf(_T(" | CERT_ERROR"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_CERT_EXPIRED) _tprintf(_T(" |
CERT_EXPIRED"));
    if (dwFlags & WEBVIEW_SECURITY_FLAG_UNSAFE) _tprintf(_T(" | UNSAFE"));

    _tprintf(_T("\n"));
}

// Query and display the current security info
static void ShowSecurityInfo(HWND hWndHost)
{
    WEBVIEW_SECURITY_INFO secInfo = { 0 };
    secInfo.dwSize = sizeof(secInfo);

    if (!WebViewGetSecurityInfo(hWndHost, &secInfo))
    {
        _tprintf(_T("WebViewGetSecurityInfo failed (Error 0x%lx)\n"),
WebViewGetLastError());
        return;
    }

    _tprintf(_T("Security Level: %s\n"), SecLevelToText(secInfo.dwLevel));
    PrintSecurityFlags(secInfo.dwFlags);

    if (secInfo.szProtocol[0])
        _tprintf(_T("Protocol: %s\n"), secInfo.szProtocol);
    else
        _tprintf(_T("Protocol: (none)\n"));

    if (secInfo.szCipher[0])
        _tprintf(_T("Cipher Suite: %s\n"), secInfo.szCipher);
    else
        _tprintf(_T("Cipher Suite: (none)\n"));

    if (secInfo.szSubject[0])
        _tprintf(_T("Certificate Subject: %s\n"), secInfo.szSubject);
    else
        _tprintf(_T("Certificate Subject: (not available)\n"));

    if (secInfo.szIssuer[0])
        _tprintf(_T("Certificate Issuer: %s\n"), secInfo.szIssuer);
    else
        _tprintf(_T("Certificate Issuer: (not available)\n"));

    PrintFileTime(_T("Valid From"), &secInfo.ftValidFrom);
```

```
    PrintFileTime(_T("Valid To "), &secInfo.ftValidTo);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetSecurityLevel, WEBVIEW_SECURITY_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetSecurityLevel Function

```
BOOL WINAPI WebViewGetSecurityLevel(
    HWND hWndHost,
    LPDWORD lpdwLevel
);
```

The **WebViewGetSecurityLevel** function returns the evaluated security level for the last navigation.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpdwLevel

A pointer to an unsigned integer which will contain the security level when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetSecurityLevel** function provides a simplified way to obtain the current security state of the page displayed in the browser. This value is an overall assessment of the page's security status and is primarily intended for use in user interface logic, such as whether to display a lock icon or warning indicator. The security level does not provide detailed diagnostic information; if your application requires additional context such as certificate details, protocol version, or specific security flags, you should call **WebViewGetSecurityInfo** instead.

The possible values returned in the ***lpdwLevel*** parameter correspond to the security level constants:

- `WEBVIEW_SECURITY_LEVEL_UNKNOWN`: The security state could not be determined.
- `WEBVIEW_SECURITY_LEVEL_INSECURE`: The page was retrieved over plain HTTP or the certificate is invalid.
- `WEBVIEW_SECURITY_LEVEL_WARNING`: The page uses weak encryption or contains mixed content.
- `WEBVIEW_SECURITY_LEVEL_NEUTRAL`: Local content or the browser is in a transitional state.
- `WEBVIEW_SECURITY_LEVEL_SECURE`: The page was retrieved using HTTPS with a valid certificate.

Because the browser engine may revise the security assessment as additional resources are loaded, the level can change over the course of a navigation. For example, a page may initially appear secure, but once mixed content is detected the level will be downgraded to `WEBVIEW_SECURITY_LEVEL_WARNING.` Applications should refresh their indicators when they receive a `WEBVIEW_EVENT_SECURITY_CHANGED` notification or after navigation has completed.

The level values are ordered, with higher values indicating stronger security. Applications should avoid upgrading an observed level within the same navigation, since subsequent checks may legitimately lower the assessment.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetSecurityInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetTimeout Function

```
INT WINAPI WebViewGetTimeout(
    HWND hWndHost
);
```

The **WebViewGetTimeout** function returns the number of seconds the client will wait for the navigation to complete.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. A value of zero specifies no timeout period has been specified.

Remarks

The **WebViewGetTimeout** function retrieves the current navigation timeout value, expressed in seconds. This value determines how long the library will wait for a navigation request to complete before it is considered to have failed due to a timeout condition.

If the timeout value is set to zero, the control uses the built-in timeouts defined by the underlying WebView2 browser engine. These internal timeouts are adaptive and depend on network conditions, DNS resolution, and connection establishment. In practice, this allows the browser to determine how long to wait before treating a navigation as unresponsive. For most applications, leaving the timeout set to zero provides behavior consistent with the Microsoft Edge browser.

If a non-zero timeout value has been set, navigation will be canceled if the specified number of seconds elapses without the page load completing. This provides a deterministic limit on how long a navigation attempt is allowed to continue before the operation fails. However, specifying a very long timeout value does not guarantee that navigation will continue for that entire duration, since the WebView2 engine may still enforce its own internal limits and terminate the request earlier.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetUserAgent Function

```
BOOL WINAPI WebViewGetUserAgent(
    HWND hWndHost,
    LPTSTR lpszUserAgent,
    INT nMaxLength,
);
```

The **WebViewGetUserAgent** function returns the current User-Agent value for this instance.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszUserAgent

A null-terminated string buffer which will contain the User-Agent string for the current session. If this parameter is NULL, the string will not be copied and the function will return the length of the string, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the *lpszUserAgent* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters in the User-Agent string, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetUserAgent** function retrieves the current User-Agent string used by this browser instance. This value is the exact string sent in the User-Agent header of outgoing HTTP requests and is determined by the underlying Microsoft Edge WebView2 runtime unless explicitly changed with **WebViewSetUserAgent**. The returned value can be used for informational purposes, such as logging the client identity or displaying diagnostic information to the user. It is also useful if your application needs to preserve the default User-Agent string before setting a custom one.

If the *lpszUserAgent* parameter is `NULL`, the function returns the length of the string in characters. This allows the caller to allocate a buffer of the appropriate size before retrieving the value. When a valid buffer is provided, the string copied to it will always be null-terminated.

The User-Agent value applies to the entire lifetime of the browser instance. Once it has been changed, all subsequent navigations will use the new string until explicitly reset or the instance is destroyed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetHeader, WebViewSetHeader, WebViewSetUserAgent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetUserDataFolder Function

```
INT WINAPI WebViewGetUserDataFolder(
    HWND hWndHost,
    LPTSTR lpszDataFolder,
    INT nMaxLength,
);
```

The **WebViewGetUserDataFolder** function returns the full path to the user data folder for the current session.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszDataFolder

A null-terminated string buffer which will contain the path to user data folder for the session. If this parameter is NULL, the string will not be copied and the function will return the length of the path, not including the terminating null character.

nMaxLength

An integer which specifies the maximum number of characters which can be copied into the string buffer. If the ***lpszDataFolder*** parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters in the User-Agent string, not including the terminating null character. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **User Data Folder** (UDF) is the WebView2 profile directory that Chromium uses to persist browsing state for this instance. It contains the browser cache, cookies, localStorage/IndexedDB data, service worker state, visited link history, site permissions, and other profile data. Each ***distinct*** folder path corresponds to a distinct browser profile. Multiple WebView2 instances that point to the same folder will share the same profile (and therefore share cookies, storage, and authentication state). Using different folder paths isolates profiles.

The user data folder is chosen when the WebView instance is created or attached via **WebViewAttachWindowEx** or **WebViewCreateWindowEx** and cannot be changed afterwards. If your application needs to switch profiles (for example, to use a different folder), you must destroy the current instance and create a new one with the desired folder.

If your application does not specify a folder or uses an empty string as the folder path, the library creates a unique per-session folder under the current user's local profile. This is an ***ephemeral profile*** that is not intended to be reused across browser instances. By default, this temporary folder is removed when the instance is detached; if the process terminates unexpectedly, the folder may remain and can be safely deleted later.

In contrast, specifying an explicit, persistent folder (for example, a subdirectory under `%LocalAppData%\WebView\YourApp`) gives you a shared profile that persists across sessions and can be reused by multiple WebView2 instances to enable single sign-on, shared cookies, and faster subsequent navigations due to caching. Use this approach when you want consistent, long-lived browser state.

The reported folder path is always an absolute, normalized local path. Environment variables in the path you supply at creation time are expanded; forward slashes are normalized to backslashes and relative names are resolved beneath the current user's local application data folder. The library rejects invalid or unsafe locations (e.g.: device names, system directories, UNC or extended path prefixes, or root drive targets) and will create intermediate directories as needed for valid paths. For multithreaded applications, you should use explicit absolute paths rather than relying on the process current directory.

Important: The user data folder is managed internally by the the WebView2 runtime, and your application should never attempt to modify, move, or delete any of the files it contains while a browser instance is active. The contents are in use by the rendering engine and interfering with them can result in data corruption or unexpected behavior. If you need to remove or relocate a profile, ensure that all browser instances using that folder have been completely closed. If you want to delete certain data, such as cookies or the browser history, use the functions provided specifically for that purpose.

You can call **WebViewGetUserDataFolder** at any time after the instance has been created to discover which profile directory is in use. Passing ***lpszDataFolder*** as `NULL` returns the required buffer length so you can allocate a buffer of the appropriate size; the returned length does not include the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGetZoomLevel Function

```
BOOL WINAPI WebViewGetZoomLevel(
    HWND hWndHost,
    INT *lpnPercent
);
```

The **WebViewGetZoomLevel** function return the current zoom level as a percentage.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpnPercent

A pointer to an integer which will contain the current zoom level as a percentage value when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGetZoomLevel** function returns the current zoom level of the browser window as an integer percentage value. A zoom level of **100** indicates that the page is being displayed at its normal size without any scaling applied. Values greater than **100** mean that the content is being magnified, and values less than **100** mean that the content is being reduced.

The zoom level is constrained by the limits defined for the control. By default, the minimum zoom level is **25** (25%) and the maximum zoom level is **400** (400%). Attempts to set values outside of this range using **WebViewSetZoomLevel** will be clamped to the supported range.

This function can be used to determine the current scaling when an application wants to preserve a user's preferred zoom level across sessions, or to query whether the content is being displayed at the default size. If the function fails, the contents of the ***lpnPercent*** parameter are undefined and should not be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewSetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGoBack Function

```
BOOL WINAPI WebViewGoBack(
    HWND hWndHost
);
```

The **WebViewGoBack** function navigates backwards through the browser history.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGoBack** function navigates the browser to the previous page in its session history, if one is available. This is equivalent to selecting the Back button in a standard web browser. If there is no previous entry in the history list, the function will succeed but no navigation will occur. The availability of backward navigation can be determined by handling the `WEBVIEW_EVENT_HISTORY_CHANGED` event and checking for the `WEBVIEW_MOVE_BACKWARD_ENABLED` flag.

Calling **WebViewClearHistory** will remove the session history for the browser instance. After the history has been cleared, calling **WebViewGoBack** will have no effect until new pages have been visited.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearHistory, WebViewGoForward, WebViewNavigateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewGoForward Function

```
BOOL WINAPI WebViewGoForward(
    HWND hWndHost
);
```

The **WebViewGoForward** function navigates forwards through the browser history.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewGoForward** function navigates the browser to the next page in its session history, if one is available. This is equivalent to selecting the Forward button in a standard web browser.

If there is no forward entry in the history list, the function will succeed but no navigation will occur. The availability of forward navigation can be determined by handling the `WEBVIEW_EVENT_HISTORY_CHANGED` event and checking for the `WEBVIEW_MOVE_FORWARD_ENABLED` flag.

Calling **WebViewClearHistory** will remove the session history for the browser instance. After the history has been cleared, calling **WebViewGoForward** will have no effect until new pages have been visited.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearHistory, WebViewGoBack, WebViewNavigateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewInEvent Function

```
BOOL WINAPI WebViewInEvent();
```

The **WebViewInEvent** function determines if the current thread is executing code within an event handler.

Parameters

None.

Return Value

If the current thread is executing code within the context of an event handler, the return value is non-zero; otherwise the return value is zero.

Remarks

The **WebViewInEvent** function checks a thread-local flag that is set whenever your code is executing inside an event handler. This allows an application to determine if the current call stack is the result of an event callback (for example, a navigation completed event or a message received event) rather than code executing on its own.

Knowing whether you are inside a event handler can be useful for several reasons:

- **Reentrancy detection**. Certain API functions should not be called while an event is still being processed, since this can cause re-entrant behavior or unexpected state changes in the underlying WebView2 control. For example, it is not permitted to attempt another navigation or reset the browser state while inside a navigation event handler.
- **Thread marshalling**. Event handlers are typically invoked on the UI thread associated with the WebView instance. By checking this function, the application can decide whether it needs to marshal an operation to the browser's thread or whether it is already executing in the correct context.
- **Application logic**. In some cases you may want to avoid triggering further navigation, script execution, or message posting in direct response to a particular event. This function lets you distinguish between code running in an event handler versus code running elsewhere in your application.

The return value from this function is only valid for the calling thread. Each thread maintains its own state, so a worker thread will always return FALSE unless it is actively executing a event callback on that thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewInitialize Function

```
BOOL WINAPI WebViewInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **WebViewInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **WebViewInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **WebViewUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **WebViewInitialize** or **WebViewUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

The calling thread must be in a single-threaded apartment (STA). During initialization the library calls **CoInitializeEx** as needed and if your process has already initialized COM as MTA (`COINIT_MULTITHREADED`), initialization will fail. This requirement comes directly from the Edge WebView2 control which cannot be created in an MTA COM application.

Before calling this function, applications may optionally call **WebViewRuntimeAvailable** to determine if the Microsoft Edge WebView2 runtime is installed and meets the minimum required version. This can be useful during application startup to provide a clear error message or guidance to the user if the runtime is missing or out of date, rather than attempting to initialize the control and handling the

failure afterward.

On modern versions of Windows, the WebView2 runtime is included as part of the operating system and is serviced through Windows Update. However, on older or unsupported versions of Windows the runtime may not be present by default and must be installed separately. In these cases, your application needs to detect the missing runtime and prompt the user to install it.

The latest WebView2 runtime version officially supported on Windows 7 and Windows 8/8.1 is version 109.0.1518.140, released January 2023. After version 109, no further updates, new features, or fixes have been provided for those operating systems. Although this library supports Windows 7 and Windows 8, these platforms are not recommended because they no longer receive security updates for the browser.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewDetachWindow, WebViewRuntimeAvailable, WebViewUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewIsVisible Function

```
BOOL WINAPI WebViewIsVisible(
    HWND hWndHost
);
```

The **WebViewIsVisible** function determines if the browser window is currently visible.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the browser window is visible, the return value is non-zero. If the window is not visible or has been closed, the return value is zero.

Remarks

The **WebViewIsVisible** function checks whether the browser window is currently visible. A return value of **TRUE** means that the browser window is being displayed. A return value of **FALSE** indicates that the window has been hidden using the **WebViewShowWindow** function or the window was closed.

This function only reports on the visibility state that was explicitly set through the API. It does not reflect other factors that might affect whether the browser content is actually visible to the user, such as the host window being minimized, overlapped by another window, or located off screen.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetChildWindow, WebViewGetFocus, WebViewRedrawWindow, WebViewSetBounds, WebViewSetFocus, WebViewShowWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewKeyEventHandled Function

```
BOOL WINAPI WebViewKeyEventHandled(
    HWND hWndHost,
    BOOL bHandled
);
```

The **WebViewKeyEventHandled** function sets an accelerator key event as handled.

Parameters

hWndHost

Handle to the window attached to the browser session.

bHandled

An integer which specifies if the accelerator key event was handled by the application. A non-zero value indicates the application has handled the accelerator key event and the WebView2 control should not process the event. A value of zero specifies that the browser should handle the accelerator key using it's normal, default behavior.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewKeyEventHandled** function is used in conjunction with the **WebViewEventProc** callback when the application receives a `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS` event. By calling this function and specifying that the key event has been handled, the application can prevent the browser from processing the key itself.

This mechanism is useful when you want to disable or override the default behavior of certain keys within the embedded browser. For example, you may wish to prevent the user from pressing **F12** to open the developer tools, block function keys that have meaning in your application, or intercept combinations such as **Ctrl+P** or **Ctrl+S** to substitute your own print or save functionality.

If *bHandled* is non-zero, the event is considered handled by the host application and WebView2 will not process the key. If *bHandled* is zero, the key is passed through and processed normally according to the browser's default behavior. This function should typically be called during the event handler for `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS`.

Note that this function only affects accelerator key events when the `WEBVIEW_OPTION_ENABLE_ACCELERATORS` option has been specified when creating or attaching the WebView. It does not apply to ordinary text input within form fields or content editable regions in the page.

Example

```
VOID CALLBACK EventHandler(HWND hWndHost, UINT uEvent, WPARAM wParam, LPARAM
lParam, DWORD_PTR dwParam)
{
    if (uEvent == WEBVIEW_EVENT_ACCELERATOR_KEYPRESS)
    {
        // wParam contains flags (WEBVIEW_ACCELERATOR_KEY_DOWN/UP/SYSTEM)
        // lParam contains the virtual-key code
        UINT vkCode = (UINT)lParam;
```

```
        if (vkCode == VK_F3) // Intercept the F3 key
        {
            // Mark the event as handled so WebView2 won't process it
            WebViewKeyEventHandled(hWndHost, TRUE);
            return;
        }
    }

    // Other events can be handled here as needed
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewNavigateUrl Function

```
BOOL WINAPI WebViewNavigateUrl(
    HWND hWndHost
    DWORD dwFlags,
    LPCTSTR lpszUrl
);
```

The **WebViewNavigateUrl** function navigates to the specified URL or local file path.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwFlags

An unsigned integer which specifies additional options for how the navigation will be performed.

Value	Description
WEBVIEW_NAVIGATE_DEFAULT	Use the default navigation behavior. The specified URL or local file path is normalized and used without additional restrictions.
WEBVIEW_NAVIGATE_SAFE_URL	Perform additional safety checks on the input before navigation. This will reject unsafe or malformed URLs, filter out potentially harmful schemes, and block traversal sequences (such as `../../`) in local paths. This option is recommended when the URL or path is provided by external or untrusted input.

lpszUrl

A null-terminated string which specifies the URL or local file to navigate to. Any leading and trailing whitespace will be removed and the URL will be normalized to ensure that it specifies a valid location. If the URL only specifies a domain name (such as `"example.com"`) the browser will default to using a secure connection. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewNavigateUrl** function accepts both standard web URLs such as `https://www.example.com/` and local file paths which are automatically normalized and converted into `file://` URLs. This allows an application to load either remote web content or content from the local filesystem.

When a local file path is specified, the function validates the path to ensure that it refers to a valid, absolute file system location and is not under restricted system folders (such as the Windows or Program Files directories). Relative paths are resolved against the current working directory, but it is recommended that applications provide absolute paths to avoid ambiguity in multithreaded or service-hosted environments.

File names and paths are normalized before navigation. Illegal or reserved device names (such as `CON`,

AUX, or **NUL**) and paths containing invalid characters will be rejected. UNC paths and extended-length paths using the **\\?** prefix are not supported. If the path points to a valid file, it will be loaded into the WebView using the appropriate **file://** URL form.

If the *dwFlags* parameter specifies **WEBVIEW_NAVIGATE_SAFE_URL**, the URL or path will undergo additional checks to ensure that it does not contain unsafe schemes, fragments, or traversal sequences. This is useful when the application accepts untrusted input that may be used for navigation.

Note that navigation to local files is supported, but applications should exercise caution when mixing remote and local content, as this may have cross-origin security implications. If your application needs to load dynamic HTML content, consider using **WebViewSetDocumentText** instead.

This function only supports navigation to standard web and local resources. Valid URL schemes are limited to **http://**, **https://**, **file://**, and **about:blank**. Other protocols or custom application schemes are not supported and will cause the function to fail. This includes the **edge://settings/profile** page or any other internal configuration settings for the Edge browser. If your application needs to handle such protocols, you should implement a custom handler to intercept those links and handle them externally.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetCurrentUrl, WebViewNormalizeUrl, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewNormalizeUrl Function

```
INT WINAPI WebViewNormalizeUrl(
   LPCTSTR lpszUrl,
   DWORD dwFlags,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **WebViewNormalizeUrl** function parses and normalizes a URL into a safe, navigable form.

Parameters

lpszUrl

A null-terminated string which specifies the URL to be normalized. This parameter cannot be NULL or an empty string.

dwFlags

Specifies one or more bit flags that control how the URL is normalized. A value of zero specifies that only standard normalization will be performed, such as removing any leading and trailing whitespace and ensuring the URL does not contain any illegal characters. Multiple flags may be combined using a bitwise or operator.

Value	Description
URL_FLAG_DEFAULT_HTTPS	When no URL scheme is provided and no scheme is otherwise required, select `https://` by default. If an explicit port is present, port 443 selects `https` and any other port selects `http`. Explicit input schemes are preserved.
URL_FLAG_REQUIRE_HTTP	When no scheme is provided, force `http://`. If the input already specifies a scheme, it is preserved.
URL_FLAG_REQUIRE_HTTPS	When no scheme is provided, force `https://`. If the input already specifies a scheme, it is preserved.
URL_FLAG_REJECT_LOCALHOST	Reject URLs that specify `localhost` as the host name.
URL_FLAG_REJECT_IPV4	Reject URLs that specify a numeric IPv4 address (for example, `192.0.2.1`) as the host.
URL_FLAG_REJECT_IPV6	Reject URLs that specify a bracketed IPv6 address (for example, `[2001:db8::1]`) as the host.
URL_FLAG_ASCII_HOSTNAME	Host names are checked to ensure that they only contain legal ASCII characters. Unicode host names are converted to ASCII (punycode) before validation, and the host name is normalized to lowercase.
URL_FLAG_ESCAPE_PATH	Percent-encode the path component as needed. Non-ASCII characters are encoded as UTF-8 and then percent-encoded per byte. Valid `%HH` sequences are preserved and upper-cased. The path is also normalized by removing duplicate slashes and collapsing `.` and `..` segments of the path.
URL_FLAG_ESCAPE_QUERY	Percent-encode the query and fragment tail as needed. The first `?` is treated as the start of the tail and characters commonly

	used in queries (such as **&**, **=**, **+**, **;**, **#**) are permitted unescaped. Non-ASCII characters are encoded as UTF-8 and then percent-encoded.
URL_FLAG_VALIDATE_UTF8	Applicable to projects using a multi-byte character set. Require the URL to use UTF-8 encoding and fail if the function encounters an invalid Unicode character. When this flag is not specified, and the URL does not use UTF-8 encoding, the function attempts to normalize the URL using the active code page.
URL_FLAG_REJECT_LOCAL_FILES	Reject absolute local and UNC paths and existing **file://** URLs. When specified, UNC paths are also rejected and unsafe file extensions are not allowed.
URL_FLAG_REJECT_UNC_PATHS	When local file normalization is permitted, reject UNC paths (for example, **\\server\share**). Files paths which use a mapped drive to a network share will be allowed.
URL_FLAG_ALLOW_UNSAFE_FILES	When converting local/UNC paths to **file://** URLs or normalizing existing **file://** URLs, permit any file extension. If this option is not specified, only a common set of safe, renderable file types are allowed. This includes HTML files, text files and common image formats.

lpszNormalized

A null-terminated string buffer which will contain the normalized URL when the function returns. This parameter may be NULL, in which case the function returns the number of characters required to store the normalized URL. If a buffer is provided, it must be large enough to contain the entire URL.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the string buffer, including the terminating null character. If this parameter is zero, the function will return the number of characters required to store the normalized URL and no characters will be copied into the buffer.

Return Value

An integer which specifies the length of the normalized URL, not including the terminating null character. A return value of zero indicates the URL could not be normalized, either due to an invalid URL or the output buffer not being large enough to store the complete URL. Call the **WebViewGetLastError** function for more information about the cause of the failure.

Remarks

The **WebViewNormalizeUrl** function parses and normalizes a candidate URL without navigating. In most cases you do not need to call this function directly because **WebViewNavigateUrl** already performs basic normalization internally before it passes the URL to the browser. Use **WebViewNormalizeUrl** when you want to enforce additional policy restrictions with the option flags, or notify users of clear validation failures before attempting navigation.

Normalization includes, depending on the flags you choose:

- Scheme defaults and enforcement. If the input omits a scheme, you can default to HTTPS (**URL_FLAG_DEFAULT_HTTPS**) or require http:// or https:// explicitly using

`URL_FLAG_REQUIRE_HTTP` or `URL_FLAG_REQUIRE_HTTPS`. If the normalizer cannot accept the scheme under your selected policy, the call fails. The same default behavior is applied with **WebViewNavigateUrl**.

- Hostname policy. You may reject localhost, numeric IPv4, or bracketed IPv6 literals, or enforce ASCII hostnames (Unicode hostnames are converted to Punycode and lower-cased). The host parsing also ensures the input isn't a Windows path accidentally pasted in place of a URL.
- Path and query normalization. Paths may be percent-encoded as needed, duplicate slashes removed, and path traversal segments are collapsed; queries and fragments may be percent-encoded while preserving common query delimiters.
- Local file handling. Absolute local and UNC paths or existing file:// URLs can be converted to a normalized file:// form unless you explicitly reject them with `URL_FLAG_REJECT_LOCAL_FILES` (and optionally the `URL_FLAG_REJECT_UNC_PATHS` option). If you intend to only permit web content, enable these reject flags. At navigation time the library also tracks whether the target is a local file.
- If your project uses the multi-byte character set instead of Unicode, and `URL_FLAG_VALIDATE_UTF8` is specified, the function will fail if the URL contains any invalid UTF-8 encoding. Without this flag, if the function detects invalid UTF-8 characters, it will attempt to validate the URL using the active code page. It is recommended that applications use UTF-8 or UTF-16 Unicode whenever possible.

This function validates and normalizes URLs but does not expand redirects, verify server reachability, or fetch headers. Use it to block undesired inputs before navigation (such as disallowing bare IPs or local files), or to present a user-friendly, fully qualified URL.

Example

```
// Match the safe normalization options and forbid local files
DWORD dwNormFlags =
    URL_FLAG_DEFAULT_HTTPS |
    URL_FLAG_ASCII_HOSTNAME |
    URL_FLAG_ESCAPE_PATH |
    URL_FLAG_ESCAPE_QUERY |
    URL_FLAG_REJECT_LOCAL_FILES |
    URL_FLAG_REJECT_UNC_PATHS;

// If normalization or navigation fails, save the last error code
DWORD dwError = NO_ERROR;

// Check how long the normalized URL will be
INT nLength = WebViewNormalizeUrl(pszUserInput, dwNormFlags, NULL, 0);
if (nLength <= 0)
    dwError = WebViewGetLastError();

if (dwError == NO_ERROR)
{
    // Allocate buffer and normalize again
    LPTSTR pszNormalizedUrl = (LPTSTR)malloc((nLength + 1) * sizeof(TCHAR));

    if (!pszNormalizedUrl)
         return; // Failed; virtual memory exhausted

    nLength = WebViewNormalizeUrl(pszUserInput, dwNormFlags, pszNormalizedUrl,
nLength + 1);
```

```
    if (nLength <= 0)
    {
        dwError = WebViewGetLastError();
    }
    else
    {
        // Navigate using the normalized URL
        if (!WebViewNavigateUrl(hWeb, WEBVIEW_NAVIGATE_DEFAULT, pszNormalizedUrl))
            dwError = WebViewGetLastError();
    }

    free(pszNormalizedUrl);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetCurrentUrl, WebViewNavigateUrl, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewPostMessage Function

```
BOOL WINAPI WebViewPostMessage (
    HWND hWndHost,
    LPCTSTR lpszMessage
);
```

The **WebViewPostMessage** function posts a string or JSON message to the current page.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszMessage

A null-terminated string which contains the message to be sent to the current page. This parameter cannot be NULL or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewPostMessage** function sends a message from the host application to the currently loaded page using the WebView2 messaging channel. The call is asynchronous: if it is invoked from a thread other than the thread that created the browser instance, the function marshals the request to the UI thread and returns immediately, with delivery performed later on that thread. Internally, this posts a private window message to the host window; when handled, it forwards the text using the `ICoreWebView2::PostWebMessageAsString` method.

For a page to receive messages from the host, web messaging must be enabled. When the `WEBVIEW_OPTION_ENABLE_MESSAGES` option is set, the wrapper enables web messages in the control settings (exposing `window.chrome.webview` to the script) and wires the corresponding message events. If this option is not enabled, the page will not see the messaging object and cannot receive messages.

On the JavaScript side, the page subscribes to messages using the standard listener pattern. Messages posted with this function are delivered as strings to the page's handler:

```
window.chrome.webview.addEventListener('message', (event) => {
 // event.data is the string posted by WebViewPostMessage
 const text = event.data;
 // If you know the string is JSON, you can parse it explicitly
 // const obj = JSON.parse(text);
});
```

Because delivery is asynchronous, there is no built-in acknowledgement and ordering is not guaranteed across navigations. If the page navigates or reloads after you call **WebViewPostMessage**, a listener registered on the previous document will not receive later messages.

Messages are Unicode strings and you should avoid sending very large payloads. For large data transfers, consider using application-level protocols rather than embedding large content in a single message. This function is primarily intended to enable your host application to notify the web page the user has taken some action which should be reflected in the browser. The web page can send

messages back to your application using `window.chrome.webview.postMessage(value)` and you can receive those messages using the **WebViewGetMessage** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewAsyncExecute, WebViewExecute, WebViewGetMessage, WebViewGetResult

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewPrintDocument Function

```
BOOL WINAPI WebViewPrintDocument(
    HWND hWndHost,
    DWORD dwOptions
);
```

The **WebViewPrintDocument** function displays the browser print dialog for the current document.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwOptions

An unsigned integer which specifies how the print dialog should be displayed.

Value	Description
WEBVIEW_PRINT_DIALOG_DEFAULT	Use Edge's print preview dialog, equivalent to the **Ctrl+P** experience. This dialog provides preview capabilities and additional print options.
WEBVIEW_PRINT_DIALOG_SYSTEM	Use the native system print dialog. This dialog allows the user to select a printer and set standard options, but does not provide a preview of the document.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewPrintDocument** function displays the print dialog for the currently loaded document. The dialog is shown asynchronously and the function will return once the print user interface has been invoked. This function does not perform the printing operation itself; it only initiates the print workflow and allows the user to select a printer, adjust preferences, and confirm printing.

This function can only be called after navigation has completed and the current document is fully loaded. If navigation is still in progress when the function is called, it will fail with an error indicating that the operation is in progress. This function only supports interactive printing through a user interface. If you need to programmatically export the document without user interaction, use the **WebViewSaveDocument** function instead.

It is recommended that most applications use the default dialog. The `WEBVIEW_PRINT_DIALOG_SYSTEM` option shows a simplified print dialog provided by browser. This is not the same dialog exposed by the Windows Common Controls API through the **PrintDlg** or **PrintDlgEx** functions, and it may differ in appearance and available options depending on the platform and version of the WebView2 runtime.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetDocumentText, WebViewGetFilteredText, WebViewNavigateUrl, WebViewSaveDocument

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewRedrawWindow Function

```
BOOL WINAPI WebViewRedrawWindow(
    HWND hWndHost
);
```

The **WebViewRedrawWindow** function forces the host window and browser control to redraw.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewRedrawWindow** function reloads the current document and forces both the host window and the embedded WebView2 child window to repaint themselves. This differs from **WebViewReloadDocument**, which simply reloads the page (similar to pressing **F5**) but does not explicitly invalidate and redraw the hosting window.

Internally, this function clears the browser cache and then forces a redraw by updating the control bounds and notifying the browser that the parent window has changed. If the host window itself is visible, it is also invalidated and updated to ensure that the client area is repainted.

Because this function performs multiple operations that can be relatively expensive, applications should generally prefer **WebViewReloadDocument** unless a complete repaint of both the control and the host window is required. For example, it can be used if rendering artifacts are visible after a resize or when switching between different DPI contexts.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCache, WebViewReloadDocument, WebViewResetBrowser

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewReloadDocument Function

```
BOOL WINAPI WebViewReloadDocument(
    HWND hWndHost,
    BOOL bClearCache
);
```

The **WebViewReloadDocument** function reloads the current document and optionally clears the cache.

Parameters

hWndHost

Handle to the window attached to the browser session.

bClearCache

An integer which indicates if the document cache for the session should be cleared. A non-zero value specifies the cache should be cleared; zero specifies the cache should not be cleared.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewReloadDocument** function refreshes the currently loaded page, similar to pressing **F5** in a web browser. If the ***bClearCache*** parameter is non-zero, the document cache is also cleared before reloading, ensuring that all resources are fetched again from the server instead of being served from the local cache.

This function is generally preferred over **WebViewRedrawWindow** when an application only needs to reload the content without forcing the host window and browser control to repaint. It provides a lightweight way to refresh the page and optionally reset cached resources, without the overhead of a full redraw of the hosting window.

An application would typically use this function in response to a user action or when it needs to ensure the latest version of a document is displayed after cached data may have become stale.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCache, WebViewRedrawWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewResetBrowser Function

```
BOOL WINAPI WebViewResetBrowser(
    HWND hWndHost
);
```

The **WebViewResetBrowser** function resets the embedded runtime control.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewResetBrowser** function clears the state of the current browser session and restores it to its default configuration. This includes discarding cached content, navigation history, and any data associated with the last document. The control is then navigated to an empty `about:blank` page and the zoom level is reset to its default value.

Internally, this function releases cached strings such as the most recent message, script result, and document text, clears the current HTTP status and security information, removes cached data, and clears the session history. A navigation to **about:blank** is performed, and the host window is invalidated to ensure that the client area is repainted. The call blocks until the navigation has completed or the operation times out.

Because this function performs a complete reset of the WebView2 control, it should be used in situations where the browser needs to be returned to a known, clean state (for example, when reinitializing a session or preparing the control for a new task). Applications that only need to refresh the current document should use **WebViewReloadDocument** or **WebViewRedrawWindow** instead, as they are lighter-weight operations.

This function cannot be called while a navigation is already in progress. Attempting to do so will fail with an error code indicating that the operation is in progress. As with other navigation-related functions, it must be called on the thread that owns the browser window, or the request will be marshaled back to the UI thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewClearCache, WebViewRedrawWindow, WebViewReloadDocument

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewRuntimeAvailable Function

```
BOOL WINAPI WebViewRuntimeAvailable();
```

The **WebViewRuntimeAvailable** function determines whether the required runtime is installed and available.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewRuntimeAvailable** function checks whether the Microsoft Edge WebView2 runtime is installed and meets the minimum required version for this library. If the runtime is available, the function returns `TRUE`; otherwise, it returns `FALSE` and sets the last error code to indicate whether the runtime is missing or is an unsupported version. If the runtime is not installed, the last error code will be set to `ST_ERROR_WEBVIEW2_NOT_INSTALLED`. If the installed runtime is older than the minimum supported version (currently `102.0.1245.22`), the last error code will be set to `ST_ERROR_INVALID_WEBVIEW2_VERSION`.

This function is one of the few API functions that can be called before **WebViewInitialize**. It allows an application to check for the availability of the WebView2 runtime at startup, and optionally provide a more helpful error message to the user before attempting to create or attach a browser control. Developers may wish to call this function during application startup or installation checks. For example, if the runtime is not present or is too old, the application can inform the user that they must install or update the Microsoft Edge WebView2 runtime in order to use features which require embedded web content.

If the function succeeds, it only guarantees that the required runtime is installed and is a sufficient version. It does not initialize or configure the runtime; that process occurs when **WebViewInitialize** is called.

Example

```
INT APIENTRY wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR
lpCmdLine, INT nCmdShow)
{
    if (!WebViewRuntimeAvailable())
    {
        DWORD dwError = WebViewGetLastError();

        if (dwError == ST_ERROR_WEBVIEW2_NOT_INSTALLED)
        {
            MessageBoxW(NULL,
                        L"The Microsoft Edge WebView2 runtime is not installed.\n"
                        L"Please install it to continue.",
                        L"MyApp", MB_ICONERROR | MB_OK);
        }
        else
        {
            MessageBoxW(NULL,
                        L"The installed Microsoft Edge WebView2 runtime is too
```

```
old.\n"
                        L"Please update to the latest version.",
                        L"MyApp", MB_ICONERROR | MB_OK);
        }

        return 0; // Exit application
    }

    // Safe to initialize the library and create a WebView2 control
    if (!WebViewInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    {
        MessageBoxW(NULL,
                    L"Unable to initialize the SocketTools WebView library",
                    L"MyApp", MB_ICONERROR | MB_OK);
        return 0;
    }

    // Create your main window, attach the browser, etc.

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetBrowserVersion, WebViewInitialize, WebViewUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSaveDocument Function

```
BOOL WINAPI WebViewSaveDocument(
    HWND hWndHost
    UINT uFileType
    LPCTSTR lpszFileName
);
```

The **WebViewSaveDocument** function saves the current document to a local file.

Parameters

hWndHost

Handle to the window attached to the browser session.

uFileType

An unsigned integer which specifies the format the document should be saved as.

Value	Description
WEBVIEW_FILE_TYPE_DEFAULT	Save using a format inferred from the file extension. If the file extension is not recognized, the contents will be saved as plain text. For example, if you specify a file name with the `.html` extension, it will be saved in HTML format. If the file name has a `.pdf` extension, it will be saved as a PDF.
WEBVIEW_FILE_TYPE_TEXT	Saves the current document as plain text. The HTML tags are removed and entities are converted to their UTF-8 equivalent. Extraneous whitespace within the text will be removed, but the overall structure of ordered and unordered lists will be preserved. Choose this format if you wish to save only the text content of a document.
WEBVIEW_FILE_TYPE_HTML	Saves the current document as a UTF-8 encoded HTML file. This includes the complete source for the page, but does not include any of the additional assets such as images. Choose this format if you wish to save the document as an editable HTML file.
WEBVIEW_FILE_TYPE_JSON	Saves the current document as a JSON formatted string. The string will be quoted and reserved characters will be escaped. Choose this format if you wish to preserve the HTML document as a JSON string for logging or embedding in a JSON object.
WEBVIEW_FILE_TYPE_MHTML	Saves the current document as a single-file web archive (MHTML) that inlines the HTML content and all of the resources it references. This format calls the DevTools `Page.captureSnapshot` method in the browser and writes the returned data to disk. MHTML is useful for offline archiving since it combines images, styles and scripts together in a single file.
WEBVIEW_FILE_TYPE_PDF	Saves the current document as a PDF file which produces

	a paginated document suitable for printing and distribution. This is functionally similar to using the browser to print to a PDF, but does not display a dialog to the user.
WEBVIEW_FILE_TYPE_JPEG	Save the current document view as a JPEG image. JPEG is a widely used image format optimized for photographs and complex images with smooth gradients. It uses lossy compression to significantly reduce file size while maintaining acceptable visual quality.
WEBVIEW_FILE_TYPE_PNG	Save the current document view as a PNG image. PNG is a lossless image format designed for crisp detail and high fidelity, especially in graphics with sharp edges, text, or flat colors. Although PNG files are typically larger than JPEGs, they preserve exact image data without quality loss.
WEBVIEW_FILE_TYPE_WEBP	Save the current document view as a WebP image. WebP is a modern image format developed by Google that supports both lossy and lossless compression, as well as transparency and animation. It delivers smaller file sizes compared to JPEG and PNG while maintaining comparable or better visual quality.

lpszFileName

A null-terminated string which specifies the name of the local file. If the path includes environment variables enclosed in percent (%) symbols, the variables will be expanded. This parameter cannot be NULL and must specify a valid file path. The file name cannot specify a device, folder name or system file. If the file is marked as read-only, the function will fail.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSaveDocument** function saves the current page to a local file in the format specified by the *uFileType* parameter. The operation is synchronous, processing messages while the browser completes the action, and only returns once the document has been created or the operation has failed. A temporary file is created during the save process and then moved into place, replacing any existing file. This ensures that an existing file will not be changed unless the browser indicates the save operation was successful. Saving is not permitted while a navigation is in progress, and attempts to do so will fail with an error.

The file name specified must be valid, and environment variables in the path are expanded before use. When `WEBVIEW_FILE_TYPE_DEFAULT` is specified, the format is inferred from the file extension. For example, `.html` or `.htm` will save the document as HTML, `.pdf` will create a PDF file and `.png` will save the page as a PNG image. If no recognized extension is present, the page will be saved as plain text by default.

When saving the current document as plain text, the HTML markup will be removed and it will be saved as a UTF-8 encoded text file. When saving as HTML, the function saves the complete HTML source for the page. The JSON format also saves the HTML markup, but encodes it as a quoted JSON

string with reserved characters escaped. Note that all text-related formats are saved using UTF-8 multi-byte encoding without a byte order mark (BOM). The text will not be automatically converted to the current user's locale. Most modern text editors automatically recognize UTF-8 encoding, but some legacy applications may assume the text is localized to the current code page.

The MHTML option creates a single-file archive which contains the page content and all of its resources such as images, style sheets, and scripts. This format is often used for offline archiving or sharing since everything is contained in one file. Saving as PDF renders the page into a paginated print-style format suitable for distribution. These formats all operate on the complete document, including content that extends beyond the visible area.

By contrast, saving as an image (JPEG, PNG, or WebP) captures only the visible viewport of the page at the time the function is called. Content that requires scrolling will not be included in the saved image. PNG is lossless and preserves transparency, while JPEG and WebP use compression to produce smaller files. WEBP generally offers better quality at a reduced size compared to JPEG, but support for the format may vary between applications. If you want a pixel-perfect representation of the current page as it appears on the screen, particularly if it contains a lot of text, it is recommended that you use the PNG format.

Because this function blocks the current thread until the save completes, it may take a noticeable amount of time to return if the current document is very large. If the function is called from a worker thread, the request is marshaled to the UI thread that manages the browser control. For applications with a graphical interface, it is recommended that you display the standard hourglass (or other busy indicator) while the save is in progress to provide a visual cue to the user.

If you want a copy of the HTML or plain text contents of the current page without saving it to a file, use the **WebViewGetDocumentText** or **WebViewGetFilteredText** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetDocumentText, WebViewGetFilteredText, WebViewNavigateUrl, WebViewPrintDocument, WebViewSetDocumentText

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetBackgroundColor Function

```
BOOL WINAPI WebViewSetBackgroundColor(
    HWND hWndHost,
    COLORREF rgbBackground
);
```

The **WebViewSetBackgroundColor** function changes the background color used behind the page content.

Parameters

hWndHost

Handle to the window attached to the browser session.

rgbBackground

An integer value which specifies the background color.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The background color specifies the fill used by the browser window when there is no content to display or the document itself does not define a background. This value only applies to the WebView host surface and does not affect page transparency. The alpha channel is ignored, and the color is treated as fully opaque.

It is important to note that this setting does not override the background defined by the currently loaded page. If the page specifies its own background color or image, that content will take precedence and may obscure the configured color. For example, setting the background to white will not force web pages to appear white unless the page itself has transparent regions.

Applications can use this function to provide a consistent appearance for empty or loading views, but should not rely on the background color as a substitute for page styling. When a browser instance is first created, the default background color is white.

A **COLORREF** is a 32-bit value used by the Windows API to represent an RGB color. The low-order byte specifies the red component, the next byte the green component, and the third byte the blue component, in the format **0x00BBGGRR**. The high-order byte is unused and should be zero. Note that **COLORREF** is specific to the Windows API and is not the same as the OLE color system (**OLE_COLOR**). OLE uses a slightly different encoding that allows system color values to be specified, which are resolved at runtime. Developers should be careful not to confuse the two.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetBackgroundColor

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetBounds Function

```
BOOL WINAPI WebViewSetBounds(
    HWND hWndHost,
    INT xPos,
    INT yPos,
    INT nWidth,
    INT nHeight
);
```

The **WebViewSetBounds** function sets the position and size of the browser child window.

Parameters

hWndHost

Handle to the window attached to the browser session.

xPos

The x-position of the child window. This is a client area coordinate of the host window, not a screen position.

yPos

The y-position of the child window. This is a client area coordinate of the host window, not a screen position.

nWidth

The width of the child window. This is expressed in device units for the host window's client area.

nHeight

The height of the child window. This is expressed in device units for the host window's client area.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetBounds** function updates the position and size of the embedded browser window relative to the client area of the host window. It calls the ***put_Bounds*** method on the underlying WebView2 controller to resize the child window that renders the web content. The specified coordinates and dimensions must fall within the client area of the host window associated with the browser instance. Attempting to specify bounds outside of the host window will result in clipping or the control being partially hidden.

The *nWidth* and *nHeight* parameters are device units, the same pixel units that the Win32 API uses with functions like **MoveWindow** and **SetWindowPos**. If the host window was created as a per-monitor DPI-aware window, the values your application gets with a `WM_SIZE` message are already scaled for the current DPI and you should just pass those values directly to this function. If your application is not DPI-aware, Windows will attempt to scale the values for you through DPI virtualization. It is important to note that these values are not logical units like Twips and they are not independent of DPI scaling.

Typically this function is called from within a `WM_SIZE` or `WM_MOVE` message handler so that the browser control resizes and repositions itself along with the parent window. You can also call this function directly to dock the browser into a specific portion of the host window's client area, such as

aligning it within a child frame or panel.

In most applications it is sufficient to call this function in response to a **WM_SIZE** message so the browser control resizes along with the client area of the host window. However, in a per-monitor DPI-aware application it may also be useful to call this function when handling **WM_MOVE**. This is because moving a top-level window between monitors can trigger a DPI change even if the window size itself does not change. Reapplying the bounds in response to **WM_MOVE** ensures the control is correctly adjusted to the new scaling factor. For applications that are not DPI-aware, handling **WM_MOVE** is generally unnecessary since the client area origin remains at (0,0) and Windows applies DPI virtualization automatically.

Calling **WebViewSetBounds** does not affect the visibility of the browser window. To hide or show the control, use the **WebViewShowWindow** function, and to query its visibility, use **WebViewIsVisible**.

Example

```
 LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
        // Attach a browser to this window as its host
        WebViewAttachWindow(hWnd, _T("https://www.example.com"), NULL, 0);
        return 0;

    case WM_SIZE:
        {
            RECT rcClient;
            GetClientRect(hWnd, &rcClient);

            // Resize the browser to occupy the full client area
            WebViewSetBounds(hWnd,
                             rcClient.left,
                             rcClient.top,
                             rcClient.right - rcClient.left,
                             rcClient.bottom - rcClient.top);
        }
        return 0;

    case WM_DESTROY:
        // Detach the browser instance from the window
        WebViewDetachWindow(hWnd);
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetFocus, WebViewIsVisible, WebViewRedrawWindow, WebViewSetFocus,

WebViewShowWindow

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetDocumentText Function

```
BOOL WINAPI WebViewSetDocumentText(
    HWND hWndHost,
    LPCTSTR lpszText
);
```

The **WebViewSetDocumentText** function replaces the current document with the specified HTML or plain text.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszText

A null-terminated string which specifies the HTML or plain text that should replace the current document. If this parameter is NULL the current document will be cleared.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetDocumentText** function replaces the currently displayed content with the specified HTML or plain text. If the text does not contain any markup, it will be wrapped in HTML as preformatted text and displayed using the default monospaced font. If the *lpszText* parameter is `NULL` or an empty string, the browser will display a blank page.

The string passed to this function is normally rendered directly by the browser. If the string is very large (over 256 KiB), the library will first write the text to a local temporary file and then instruct the browser to navigate to that file. This is done to work around limitations in the Edge WebView2 control when handling long inline strings. The temporary file will be automatically deleted when it is no longer needed.

There are two versions of this function, one that accepts 8-bit text (**WebViewSetDocumentTextA**) and one that accepts 16-bit Unicode text (**WebViewSetDocumentTextW**). The correct version of the function is called based on your project settings and whether the `UNICODE` macro is defined. If you call the ANSI version of this function, it expects the text will be UTF-8 encoded. If the text contains anything other than ASCII characters, it is recommended that you use the Unicode version of this function.

Important: It is possible to embed markup that references images, scripts, or other resources located on the Internet. While this is supported, you should be aware of potential risks when combining local content with external resources. This can create mixed-content scenarios and introduce cross-origin or security issues. Applications that render untrusted content should avoid linking to remote resources and ensure that the HTML provided is properly validated.

Example

In this example, the function replaces the current page with a basic HTML document. The `<img>` tag references an external image from a website. This is supported, but as noted above, applications should carefully consider the security implications of mixing local content with external resources.

```
// Replace the current document with a simple HTML page
```

```
LPCTSTR pszHtml =
    _T("<html><head><title>Hello</title></head>")
    _T("<body><h1>Hello, world!</h1>")
    _T("<p>This content was created dynamically.</p>")
    _T("<img src=\"https://example.com/logo.png\" alt=\"Logo\"/>")
    _T("</body></html>");

if (!WebViewSetDocumentText(hWndHost, pszHtml))
{
    DWORD dwError = WebViewGetLastError();
    // Handle the error
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetDocumentText, WebViewGetDocumentTitle, WebViewGetFilteredText, WebViewNavigateUrl

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetDownloadPath Function

```
BOOL WINAPI WebViewSetDownloadPath(
    HWND hWndHost,
    DWORD dwFileId,
    LPCTSTR lpszFilePath
);
```

The **WebViewSetDownloadPath** function changes the location of the local file for the specified download.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwFileId

An unsigned integer value which uniquely identifies the download. This value cannot be zero.

lpszFilePath

A null-terminated string which specifies the new path for the downloaded file. This value may be a `file://` URL or it may specify a local file path. If the path contains environment variables surrounded with percent (%) symbols, the variables will be expanded. This parameter cannot be NULL or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The specified path may be provided as either a file:// URL or a local file system path. If a file:// URL is given, it will be converted to a local path before use. Environment variables enclosed in percent signs (such as `%TEMP%`) are expanded automatically. If the path refers to an existing directory rather than a file name, the original file's base name is preserved and the file will be created in that directory. This allows applications to easily redirect a download into a specific folder without changing the file name.

The function always normalizes the provided path. If a relative path is specified, it is resolved against the current working directory of the process. Because this can lead to unpredictable results in multithreaded applications, it is strongly recommended that applications always provide absolute paths. By default, downloaded files are typically stored in the current user's Downloads folder.

Several restrictions are enforced on the path to help ensure safety and compatibility. Device names such as `CON`, `PRN`, or `NUL` are not permitted. Downloads cannot be redirected into system folders such as the Windows folder or Program File folder. The path cannot contain reserved characters, and files that are marked as read-only will not be overwritten. If any of these conditions are detected, the function will fail with an error.

This function can only be called within a handler for the `WEBVIEW_EVENT_DOWNLOAD_STARTING` event. Once the download has begun, the local file path cannot be changed. If you wish to move the file after it has been downloaded, you can use the **WebViewGetDownloadPath** function to obtain its current location and then use a function like **MoveFileEx** to move the file to a new folder.

Note that you should avoid showing modal dialogs such as the common "Save File" dialog within event handlers. Because the browser events are raised within the context of the WebView2 UI thread,

displaying a dialog asking the user for a new download location can prevent the control from processing its internal message loop, potentially leading to stalled downloads.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewCancelDownload, WebViewEnumDownloads, WebViewGetDownloadInfo, WebViewGetDownloadPath

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetFocus Function

```
BOOL WINAPI WebViewSetFocus(
    HWND hWndHost
);
```

The **WebViewSetFocus** function changes focus to the browser control.

Parameters

hWndHost

Handle to the window attached to the browser session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetFocus** function gives input focus to the embedded browser control. Once focus has been set, keyboard input such as typing or navigation keys will be directed to the content displayed by the control rather than to other controls in the host window. This is equivalent to the user clicking inside the browser area.

Setting focus does not make the host window itself the active foreground window. If the host window is not active, input will still not be directed to the browser until the window is activated by the user or programmatically. Applications that need to determine whether the browser currently has focus can call **WebViewGetFocus**.

Example

```
BOOL bHasFocus = FALSE;

if (WebViewGetFocus(hWndHost, &bHasFocus))
{
    if (!bHasFocus)
    {
        // Give the browser control focus if it does not have it
        WebViewSetFocus(hWndHost);
    }
}
else
{
    // Handle error
    DWORD dwError = WebViewGetLastError();
    _tprintf(_T("WebViewGetFocus failed, error=0x%lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetFocus, WebViewGetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetHeader Function

```
BOOL WINAPI WebViewSetHeader (
   HWND hWndHost,
   LPCTSTR lpszHeaderName,
   LPCTSTR lpszHeaderValue
);
```

The **WebViewSetHeader** function sets a request header value to be included with subsequent navigations.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszHeaderName

A null-terminated string which specifies the name of the request header value to set or change.

lpszHeaderValue

A null-terminated string which specifies the value of the header. If this value is NULL or an empty string, the request header will be removed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetHeader** function adds or updates a custom request header that will be included with subsequent navigations. Request header names are not case-sensitive, and any leading or trailing whitespace is ignored. If the specified header value is NULL or an empty string, the header will be removed from the current list of custom headers. These changes only affect future requests; they do not modify headers already sent for the current navigation.

Certain standard headers are controlled by the underlying WebView2 control and may be ignored if set programmatically. Examples include `Host`, `Referer`, `Origin`, `User-Agent`, and most authentication headers. Attempting to override these may have no effect. For headers that can be set, the specified values will be merged with the default request headers used by WebView2 when making requests. If you wish to change the `User-Agent` header value, use the **WebViewSetUserAgent** function.

When defining custom request headers, it is recommended to use names prefixed with X- (for example, `X-Application-ID` or `X-Request-Token`). This convention helps distinguish application-specific headers from those defined by the HTTP standard, and reduces the likelihood of conflicts with headers that are managed internally by WebView2 or intermediaries such as proxies and gateways. While most modern systems will accept arbitrary header names, following this convention improves compatibility and avoids unexpected behavior.

If you need to clear all custom request headers that have been set, call **WebViewClearHeaders** with the `WEBVIEW_HEADERS_REQUEST` option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h

Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewClearHeaders, WebViewGetFirstHeader, WebViewGetHeader, WebViewGetNextHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetIcon Function

```
BOOL WINAPI WebViewSetIcon(
   HWND hWndHost,
   HINSTANCE hResInst,
   LPCTSTR lpszIconName,
   UINT uResId,
   DWORD dwFlags
);
```

The **WebViewSetIcon** function sets custom large and small icons for the host window.

Parameters

hWndHost

Handle to the window attached to the browser session.

hResInst

Handle to the module instance that contains the icon resource. If this parameter is NULL, the ***lpszIconName*** parameter should specify the path to an icon file.

lpszIconName

A null-terminated string that specifies the name of the icon resource or a path to an icon file. If this parameter is not NULL, it takes precedence over ***uResId***.

uResId

An unsigned integer which specifies the numeric identifier of the icon resource within the module. This is used when ***lpszIconName*** is NULL.

dwFlags

An unsigned integer which specifies which icons to update. This must be one or more of the values shown in the following table.

Value	Description
WEBVIEW_ICON_BIG	Update the large window icon.
WEBVIEW_ICON_SMALL	Update the small window icon.
WEBVIEW_ICON_BOTH	Update both large and small icons.
WEBVIEW_ICON_DPI_AWARE	Choose the system-recommended small icon size for the current DPI when possible.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetIcon** function updates the large and/or small icons associated with the host window created using the **WebViewCreateWindow** or **WebViewCreateWindowEx** function. This is equivalent to sending the `WM_SETICON` message to the host window, but automatically handles both the large and small icons and supports DPI-aware sizing when requested.

If the ***hRetInst*** parameter is not NULL, either a string resource name (***lpszIconName***) or a numeric resource identifier (***uResId***) can be specified to identify the icon. If both are provided, the string

resource name takes precedence. The resource must exist in the module instance specified by *hResInst*. If neither *lpszIconName* nor *uResId* resolves to a valid icon, the function fails.

if the *hResInst* parameter is NULL, the *lpszIconName* parameter must specify the name of a file which contains the icon and the *uResId* parameter is ignored. It is recommended that you provide an absolute path to the icon file. If the path includes environment variables surrounded with percent (%) symbols, those variables will be automatically expanded. The file must use the standard Windows icon format.

The *dwFlags* parameter determines which icons are updated. You can set the large icon (used in the Alt-Tab interface and title bar), the small icon (shown in the window caption and taskbar), or both. If `WEBVIEW_ICON_DPI_AWARE` is specified, the system will choose an appropriate small icon size based on the current display scaling.

Windows icons are stored in the standard `.ico` file format, which can contain one or more images at different resolutions and color depths. When you compile icons into your application's resources and reference them by name or identifier, Windows will automatically select the most appropriate image for the current display settings. This allows a single icon resource to provide 16x16, 32x32, 48x48, or higher-resolution images within the same file.

The operating system chooses which image to use based on context and DPI scaling. For example, a 16x16 version may be displayed in a window caption or taskbar, while a 32x32 or 48x48 version may be shown in the Alt-Tab interface or Windows Explorer. If `WEBVIEW_ICON_DPI_AWARE` is specified, the system can pick a small-icon size that best matches the current DPI rather than always defaulting to 16x16.

Calling this function does not destroy the icon resource itself; ownership remains with the module that loaded it. If you dynamically create icons at runtime using **CreateIcon** or **LoadImage**, you are responsible for releasing them with **DestroyIcon** if they are no longer needed.

This function can only be used with windows created by this library, it cannot be used with windows you attach the browser to using the **WebViewAttachWindow** or **WebViewAttachWindowEx** functions.

Example

```
// Define the path to the application's icon file
LPCTSTR pszIconFile = _T("%LocalAppData%\\MyApp\\MyApp.ico");

HWND hWnd = WebViewCreateWindow(_T("about:blank"), NULL, 0);
if (hWnd)
{
    // Set both large and small icons for the WebView host window
    WebViewSetIcon(
        hWnd,
        NULL,
        pszIconFile,
        0,
        WEBVIEW_ICON_BOTH | WEBVIEW_ICON_DPI_AWARE
    );
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewCreateWindow, WebViewCreateWindowEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetLastError Function

```
VOID WINAPI WebViewSetLastError(
    DWORD dwErrorCode
);
```

The **WebViewSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE or NULL. Those functions which call **WebViewSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **WebViewGetLastError** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetErrorString, WebViewGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetOption Function

```
BOOL WINAPI WebViewSetOption(
    HWND hWndHost,
    DWORD dwOption,
    BOOL bEnable
);
```

The **WebViewSetOption** function enables or disables a specific option.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwOption

An unsigned integer which specifies the option to be changed. Refer to the WebView Options documentation for the list of available options. Only a single option may be specified; you cannot combine multiple option bitflags when calling this function.

bEnable

An integer value which specifies if the option should be enabled. If this value is non-zero, the option is enabled. If the value is zero, the option is disabled.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetOption** function enables or disables a single option for the browser instance attached to the specified host window. Each option corresponds to a specific feature or behavior such as scripting, zoom controls, developer tools, or privacy settings. Only one option bitflag may be specified in the ***dwOption*** parameter when calling this function. It is not valid to combine multiple option flags with a bitwise OR. If you need to change the state of multiple options, call this function once for each option you want to modify.

Some options can be changed dynamically at runtime, while others only take effect during creation of the browser control. For example, options related to scripting, dialogs, or developer tools can typically be toggled while the control is active. In contrast, options that affect the user data folder, profile isolation, or privacy mode must be set at creation time using **WebViewAttachWindowEx** or **WebViewCreateWindowEx**. Attempting to change those options will fail once the browser control has been created.

If the function succeeds, the option state will be updated immediately. The application can verify the new state by calling **WebViewGetOption** or by retrieving all current options with **WebViewGetAllOptions**. If the function fails, no changes are made and extended error information can be obtained with **WebViewGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewGetAllOptions, WebViewGetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetTimeout Function

```
INT WINAPI WebViewSetTimeout(
    HWND hWndHost,
    INT nTimeout
);
```

The **WebViewSetTimeout** function sets the number of seconds the browser will wait for a response from the server.

Parameters

hWndHost

Handle to the window attached to the browser session.

nTimeout

The number of seconds to wait for the navigation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WEBVIEW_ERROR. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetTimeout** function specifies the maximum number of seconds that the library should wait for a navigation request to complete. This timeout begins when navigation starts and ends when the navigation completes or fails.

If the timeout period is set to zero, the library does not enforce its own timer and instead relies on the default timeouts implemented by the WebView2 engine. These built-in timeouts automatically handle conditions such as slow DNS resolution, stalled connections, or unreachable servers, and generally behave in the same way as the Microsoft Edge browser.

If a non-zero timeout period is specified, the control will cancel the navigation if the internal timer exceeds that value and the page has not finished loading. When this occurs, the navigation is stopped and a timeout error is reported through the standard navigation event notifications. This allows applications to fail fast in cases where a site is not responding, rather than waiting for the browser's adaptive timeouts to expire.

Note that specifying a very large timeout value will not force the browser to wait indefinitely. The WebView2 engine may still apply its own internal maximums and terminate the navigation earlier if it decides the request cannot succeed. Applications should select timeout values that balance responsiveness and reliability.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindowEx, WebViewCreateWindowEx, WebViewGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetUserAgent Function

```
BOOL WINAPI WebViewSetUserAgent(
    HWND hWndHost,
    LPCTSTR lpszUserAgent
);
```

The **WebViewSetUserAgent** function changes the default User-Agent string for this instance.

Parameters

hWndHost

Handle to the window attached to the browser session.

lpszUserAgent

A null-terminated string which specifies the new User-Agent header value for subsequent requests. If this parameter is NULL or an empty string, the User-Agent header will revert back to the default value used by the Edge WebView2 control.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetUserAgent** function sets the string that will be sent in the User-Agent header for all subsequent HTTP requests made by this browser instance. This function should always be used to modify the User-Agent header value; you should not set the header directly using the **WebViewSetHeader** function.

Applications may want to override the default User-Agent in several scenarios:

- Requesting a mobile or tablet version of a site by providing a User-Agent string that identifies as a mobile browser.
- Emulating a different browser for testing purposes.
- Including custom product tokens identifying the host application to the server.

The general format of a User-Agent string follows the conventions described in RFC 9110:

```
User-Agent: product / version (comment) [ product / version ... ]
```

For example, a typical User-Agent string might look like:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/140.0.0.0 Safari/537.36 Edg/140.0.0.0
```

This value is global to the instance. Once changed, it will remain in effect until explicitly reset or until the browser window is destroyed. When setting a custom User-Agent value, developers should ensure it is a syntactically valid string. Some websites may reject requests or provide degraded functionality if the value is malformed or omits key tokens.

If you want to change the User-Agent string to identify your application, it is recommended that you append your application name and version to the default value.

Example

```
// First, get the length of the current User-Agent string
```

```
INT nLength = WebViewGetUserAgent(hWndHost, NULL, 0);
if (nLength <= 0)
{
    _tprintf(_T("Unable to get User-Agent string (error 0x%lx)\n"),
WebViewGetLastError());
    return;
}

// Allocate a buffer large enough to append the application name and version;
// note the leading space in the name to separate it from other tokens
const TCHAR szAppId[] = _T(" MyApp/1.0");

INT nTotalLength = nLength + (INT)_tcslen(szAppId) + 1;
LPTSTR lpszUserAgent = (LPTSTR)malloc(nTotalLength * sizeof(TCHAR));

if (!lpszUserAgent)
    return;

// Retrieve the current User-Agent value from the browser
if (WebViewGetUserAgent(hWndHost, lpszUserAgent, nTotalLength) <= 0)
{
    _tprintf(_T("Failed to copy User-Agent string (error 0x%lx)\n"),
WebViewGetLastError());
    free(lpszUserAgent);
    return;
}

// Append our application identifier to the User-Agent string
_tcscat_s(lpszUserAgent, nTotalLength, szAppId);

// Update the User-Agent header with the new value
if (!WebViewSetUserAgent(hWndHost, lpszUserAgent))
{
    _tprintf(_T("Failed to update the User-Agent string (error 0x%lx)\n"),
WebViewGetLastError());
}
else
{
    _tprintf(_T("User-Agent successfully updated to:\n%s\n"), lpszUserAgent);
}

free(lpszUserAgent);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetHeader, WebViewGetUserAgent, WebViewSetHeader

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewSetZoomLevel Function

```
BOOL WINAPI WebViewSetZoomLevel(
    HWND hWndHost,
    INT nPercent
);
```

The **WebViewSetZoomLevel** function changes the current browser zoom level.

Parameters

hWndHost

Handle to the window attached to the browser session.

nPercent

An integer which specifies the zoom level as a percentage. The minimum zoom level is 25 and the maximum is 400.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewSetZoomLevel** function changes the zoom level of the current browser window to the specified percentage. A zoom level of **100** represents the default size with no scaling applied. Larger values magnify the content, while smaller values reduce it.

The zoom percentage is constrained by the limits defined for the control. The minimum supported zoom level is **25** (25%) and the maximum is **400** (400%). If a value outside of this range is specified, it will be clamped to the nearest valid zoom level.

Changing the zoom level affects how the page content is displayed but does not alter the underlying document. This is equivalent to adjusting the zoom in a standard web browser. The new zoom level applies to the current browser instance and will remain in effect until it is changed again, or the instance is destroyed.

If the function fails, the zoom level remains unchanged. To query the current zoom level, call **WebViewGetZoomLevel**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetZoomLevel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewShowWindow Function

```
BOOL WINAPI WebViewShowWindow(
    HWND hWndHost,
    BOOL bVisible
);
```

The **WebViewShowWindow** function shows or hides the browser window.

Parameters

hWndHost

Handle to the window attached to the browser session.

bVisible

An integer value which specifies if the browser window should be displayed. If this value is non-zero the browser will be shown. If the value is zero, the browser window will be hidden.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewShowWindow** function changes the visibility state of the browser control. When the browser was created as a top-level window using **WebViewCreateWindow** or **WebViewCreateWindowEx**, this function internally calls the standard **ShowWindow** API to show or hide the entire browser window. The window will be restored or hidden according to the value of the *bVisible* parameter.

When the browser is embedded in an existing host window using **WebViewAttachWindow** or **WebViewAttachWindowEx**, the function does not change the visibility of the parent window. Instead, it sends an internal message to the control to show or hide the child window which contains the WebView2 browser. This enables the host application to toggle the visibility of the browser control independently of other controls in the same window.

This function does not destroy the browser instance or unload its current content. Hiding the browser simply makes it invisible; it can be shown again without requiring a reload of the current document. The current visibility state can be queried using the **WebViewIsVisible** function.

Hiding the browser window can also provide performance benefits in scenarios where the application does not need to display the page contents for a period of time. While background tasks such as script execution and network activity continue as normal, the browser will not spend resources on rendering or updating the visual output when the control is not visible. This can reduce CPU and GPU usage, particularly when the page is dynamic or graphically intensive.

Calling this function has no effect if the handle passed in *hWndHost* is not currently associated with an active browser session. If the window is minimized or closed, the function will not restore it; it only changes the visible state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewGetFocus, WebViewIsVisible, WebViewRedrawWindow, WebViewSetBounds, WebViewSetFocus

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewUninitialize Function

```
VOID WINAPI WebViewUninitialize();
```

The **WebViewUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **WebViewInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **WebViewUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages.

There should be a call to **WebViewUninitialize** for every successful call to **WebViewInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated when the reference count for the library is zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewDetachWindow, WebViewResetBrowser, WebViewInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewWaitForEvent Function

```
BOOL WINAPI WebViewWaitForEvent(
    HWND hWndHost
    UINT uEventId,
    DWORD dwMilliseconds
);
```

The **WebViewWaitForEvent** function waits until a specific event occurs or the timeout period expires.

Parameters

hWndHost

Handle to the window attached to the browser session.

uEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the WebView Events documentation.

dwMilliseconds

The number of milliseconds to wait for the event to occur. If this amount of time elapses, the function returns. If this value is zero, the function will return successfully if the last event handled matches the ***uEventId*** parameter. This can be used to effectively poll the browser to determine if the specified event has occurred. This value may be **INFINITE** in which case the function will not return until the event occurs.

Return Value

If the function succeeds, the return value is non-zero. If the function fails or the timeout period is exceeded, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewWaitForEvent** function suspends the calling thread until the specified event is raised or the timeout interval expires. While waiting, this function continues to process the thread's Windows message queue. This ensures that the user interface remains responsive and that event callbacks are able to execute on the correct thread.

Because messages are dispatched while this function blocks, application code must be written with the possibility of reentrancy in mind. That is, another message handler or event callback may run before **WebViewWaitForEvent** returns. If your application maintains shared state, you should use appropriate synchronization (critical sections, mutexes, etc.) to ensure it remains consistent.

This function should generally be called from the same thread that created the browser instance. Calling it from a worker thread that does not own a message loop may prevent events from being delivered. If you require a fully non-blocking design, you should use the event callback function specified when creating or attaching the browser window, rather than waiting synchronously.

You should not call **WebViewWaitForEvent** from within an event handler. Doing so can result in unexpected behavior, as the function will attempt to wait for additional events and dispatch messages while another event is already being processed. This may lead to subtle reentrancy issues or even deadlocks depending on the event ordering. Event handlers should return promptly, and synchronous waits should be performed only outside of callback contexts.

Note that if ***dwMilliseconds*** is set to **INFINITE**, the function will not return until the event occurs. To avoid unresponsive behavior, it is recommended you use a finite timeout value unless you are certain

the event will always be signaled within a reasonable period.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewCreateWindow, WebViewEventProc, WebViewWaitUntilClosed

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebViewWaitUntilClosed Function

```
BOOL WINAPI WebViewWaitUntilClosed(
    HWND hWndHost,
    DWORD dwMilliseconds
);
```

The **WebViewWaitUntilClosed** function waits until the host window is closed or destroyed.

Parameters

hWndHost

Handle to the window attached to the browser session.

dwMilliseconds

The number of milliseconds to wait for the browser window to close. If this amount of time elapses, the function returns. If this value is zero, the function will not enter a wait state and returns immediately. This value may be `INFINITE` in which case the function will not return until the window is closed.

Return Value

If the browser window has been closed, the return value is non-zero. If the timeout period has been exceeded or the function fails, the return value is zero. To get extended error information, call **WebViewGetLastError**.

Remarks

The **WebViewWaitUntilClosed** function blocks the calling thread until the browser window created by **WebViewCreateWindow** or **WebViewCreateWindowEx** has been closed or destroyed, or until the specified timeout interval has elapsed. Internally, this function waits on the thread handle associated with the WebView2 instance using the `WaitForSingleObject` function.

This function can only be used with browser windows that were created by this API. It is not valid to call this function with a window handle that was attached using **WebViewAttachWindow** or **WebViewAttachWindowEx**. Attempting to do so will cause the function to fail.

Unlike **WebViewWaitForEvent**, this function does not process any Windows messages while it is waiting. As a result, it should not be used by applications that have a message loop, such as GUI programs that create windows with menus, controls, or dialog boxes. Because messages are not dispatched, the application's user interface may become unresponsive if this function is called from the primary UI thread.

This function is intended primarily for use in console applications or background worker threads, where it may be necessary to determine when the browser window has been closed by the user. GUI applications should use the **WebViewEventProc** callback to detect the `WEBVIEW_EVENT_WINDOW_DETACHED` event instead of calling this function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cshtmv12.lib

See Also

WebViewAttachWindow, WebViewCreateWindowEx, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Data Structures

- WEBVIEW_DOWNLOAD_INFO
- WEBVIEW_SECURITY_INFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEBVIEW_DOWNLOAD_INFO

The **WEBVIEW_DOWNLOAD_INFO** structure contains information about the current state of a file download.

```
typedef struct _WEBVIEW_DOWNLOAD_INFO
{
    DWORD dwSize;
    DWORD dwFileId;
    DWORD dwStatus;
    ULARGE_INTEGER uiBytesTotal;
    ULARGE_INTEGER uiBytesCopied;
    DWORD dwTimeElapsed;
    TCHAR szFileName[260];
    TCHAR szFileUrl[2048];
} WEBVIEW_DOWNLOAD_INFO, *LPWEBVIEW_DOWNLOAD_INFO;
```

Members

dwSize

Specifies the size of the structure in bytes. This member must be initialized by the caller.

dwFileId

A unique identifier assigned to the download. This value can be used with functions such as **WebViewGetDownloadInfo** and **WebViewCancelDownload** to query or manage the download.

dwStatus

Indicates the current state of the download. This will be one of the following values:

Name	Description
WEBVIEW_DOWNLOAD_STARTED	The file download has started and is in-progress.
WEBVIEW_DOWNLOAD_COMPLETED	The file download has completed successfully.
WEBVIEW_DOWNLOAD_CANCELED	The file download has been canceled.
WEBVIEW_DOWNLOAD_FAILED	The file download failed due to a network error or other problem.

uiBytesTotal

The total size of the file, in bytes, as reported by the server. If the server did not provide the file size, this value may be zero.

uiBytesCopied

The number of bytes that have been downloaded so far. If the download has completed, this value will be equal to ***uiBytesTotal***. If the total size is unknown, this value still reflects the number of bytes received.

dwTimeElapsed

The elapsed time for the download, in milliseconds. For completed downloads this is the total time from start to finish. For active downloads this is the time since the download began. If the download has not started, this value is zero.

szFileName

A null-terminated string which specifies the local file name associated with the download. If the application has not changed the file path using **WebViewSetDownloadPath**, this will be the

default file name assigned by the browser. Typically the file will be stored in the current user's Downloads folder.

szFileUrl

A null-terminated string which specifies the URL of the file being downloaded. This is the fully qualified resource location provided to the browser.

Remarks

This structure provides detailed information about the progress and final state of a download managed by the browser. Before calling a function that populates this structure, such as **WebViewGetDownloadInfo**, the caller must set the ***dwSize*** member to the size of the structure. This ensures that applications remain compatible if new fields are added in future updates.

The ***uiBytesTotal*** and ***uiBytesCopied*** members allow applications to report download progress or log completed downloads. If the total file size is not known, ***uiBytesTotal*** will be zero and applications should base their progress reporting on the number of bytes copied alone.

The ***dwFileId*** member can be obtained from **WebViewEnumDownloads**, and is required to retrieve download information or to cancel a pending download.

Both ANSI (A) and Unicode (W) versions of this structure are provided. Applications that call **WebViewGetDownloadInfoW** will populate the wide-character members of the structure (such as ***szFileName*** and ***szFileUrl***) with UTF-16 encoded text. Applications that call **WebViewGetDownloadInfoA** will receive the ANSI version of the structure, where those same members are returned as UTF-8 encoded strings. The default **WEBVIEW_DOWNLOAD_INFO** macro maps to one or the other depending on whether the UNICODE preprocessor symbol is defined at compile time.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewCancelDownload, WebViewEnumDownloads, WebViewGetDownloadInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WEBVIEW_SECURITY_INFO

The **WEBVIEW_SECURITY_INFO** structure contains information about the current state of a secure connection.

```
typedef struct _WEBVIEW_SECURITY_INFO
{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwLevel;
    TCHAR szProtocol[32];
    TCHAR szCipher[64];
    TCHAR szSubject[128];
    TCHAR szIssuer[128];
    FILETIME ftValidFrom;
    FILETIME ftValidTo;
} WEBVIEW_SECURITY_INFO, *LPWEBVIEW_SECURITY_INFO;
```

Members

dwSize

Specifies the size of the structure in bytes. This member must be initialized by the caller.

dwFlags

A bitmask that provides additional detail about the connection and certificate state. One or more values may be set to indicate conditions such as unencrypted content, use of weak algorithms, or general certificate errors. It may be one or more of the security flags listed below.

Name	Description
WEBVIEW_SECURITY_FLAG_NONE	No additional security indicators.
WEBVIEW_SECURITY_FLAG_UNENCRYPTED	Content is not encrypted.
WEBVIEW_SECURITY_FLAG_INSECURE	Site is insecure using outdated protocols or weak ciphers.
WEBVIEW_SECURITY_FLAG_CERT_WEAK	The certificate uses weak signatures.
WEBVIEW_SECURITY_FLAG_CERT_ERROR	The certificate is invalid. This is a general error condition.
WEBVIEW_SECURITY_FLAG_CERT_EXPIRED	The certificate expired but is otherwise valid.
WEBVIEW_SECURITY_FLAG_UNSAFE	The site is considered potentially unsafe (it failed reputation checks).

dwLevel

Overall security level for the current page. This is a summarized assessment intended for UI decisions, such as showing a lock icon or warning. It maps to one of the security level values listed below.

Name	Description
WEBVIEW_SECURITY_LEVEL_UNKNOWN	The security state could not be determined.
WEBVIEW_SECURITY_LEVEL_INSECURE	The page was retrieved using plain HTTP or the site has an invalid certificate.

WEBVIEW_SECURITY_LEVEL_WARNING	The page contains mixed content or only weak encryption is available.
WEBVIEW_SECURITY_LEVEL_NEUTRAL	Local content or the browser is in a transitional/loading state.
WEBVIEW_SECURITY_LEVEL_SECURE	The page was retrieved using HTTPS and the site has a valid certificate.

szProtocol

A null-terminated string that names the negotiated security protocol, such as `"TLS 1.3"` or `"TLS 1.2"`. For plain HTTP and local content, this string is empty.

szCipher

A null-terminated string that identifies the selected cipher suite, such as `"TLS_AES_128_GCM_SHA256"`. This string is empty if the site is not secure or local content is being displayed.

szSubject

A null-terminated string which contains the certificate subject. Typically this includes the common name (CN) and/or subject alternative name which matches the hostname.

szIssuer

A null-terminated string which specifies the certificate issuer. This is normally the name of the Certificate Authority (CA) which provided the certificate.

ftValidFrom

The date and time when the certificate was issued. This is represented as a standard FILETIME structure.

ftValidTo

The date and time when the certificate expires. This is represented as a standard FILETIME structure.

Remarks

The security information reflects the state of the last navigation. During navigation, values can be provisional; the final, authoritative values are available after the navigation completes. The ***dwLevel*** value is an assessment that may be adjusted by the engine for conditions such as mixed content, weak ciphers, or certificate problems. The ***dwFlags*** field provides the underlying reasons and may have multiple bitflags set. Applications should check for combinations (for example, a page might be secure overall but also report a warning flag if subresources were blocked).

For plain HTTP (unencrypted) pages, ***dwLevel*** is typically `WEBVIEW_SECURITY_LEVEL_INSECURE`. The ***szProtocol*** and ***szCipher*** members are empty strings and ***dwFlags*** will include `WEBVIEW_SECURITY_FLAG_UNENCRYPTED`. For HTTPS with a valid certificate, ***dwLevel*** is typically `WEBVIEW_SECURITY_LEVEL_SECURE`, and other members identify the negotiated TLS version and cipher suite. If the certificate is expired or otherwise invalid, the security level is downgraded and the corresponding certificate flags are set.

The ***szSubject*** and ***szIssuer*** strings are for display and logging purposes; certificate parsing and validation is handled by the browser. The validity timestamps (***ftValidFrom*** and ***ftValidTo***) are provided to enable simple checks, but applications do not need to implement certificate validation logic themselves.

Set the ***dwSize*** member to the actual size of the structure before calling **WebViewGetSecurityInfo**. If the caller passes a different size than expected, the function will fail or only populate the fields known to that size.

Both ANSI (`A`) and Unicode (`W`) variants are provided. When compiling with `UNICODE` defined, the `TCHAR` strings are UTF-16. In ANSI builds, they are UTF-8. The **WebViewGetSecurityInfo** macro resolves to the appropriate version based on the project's character set.

Because the browser can update security state as additional resources are loaded, applications that display a security indicator should refresh it on navigation completion events. Treat the security level as ordered and do not "upgrade" a previously observed state during the same navigation unless a later callback provides a higher confidence result. Note that it's also possible for the security level to be `WEBVIEW_SECURITY_LEVEL_SECURE` during navigation, but may be downgraded to `WEBVIEW_SECURITY_LEVEL_WARNING` when navigation completes due to mixed content in the page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h.
Unicode: Implemented as Unicode and ANSI versions

See Also

WebViewGetSecurityInfo

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

CWebView Constants

Name	Value	Description
Options		
WEBVIEW_OPTION_ENABLE_NOTHING	0	No options enabled. This is very restrictive and will not permit the use of JavaScript.
WEBVIEW_OPTION_ENABLE_ACCELERATORS	0x00001	Enables common keyboard accelerators (e.g., Ctrl+L, F5). Accelerator key events are surfaced via `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS`.
WEBVIEW_OPTION_ENABLE_CONTEXT_MENU	0x00002	Shows the built-in right-click context menu. Clear this to fully suppress the menu.
WEBVIEW_OPTION_ENABLE_STATUSBAR	0x00004	Displays status text (e.g., link targets, loading state) in the bottom-left status area.
WEBVIEW_OPTION_ENABLE_ZOOM_CONTROL	0x00008	Allows user zoom (Ctrl+Plus/Minus, Ctrl+mouse-wheel). If disabled, zoom remains fixed unless set by API.
WEBVIEW_OPTION_ENABLE_AUDIO	0x00010	Permits audio playback in the WebView. Disable to mute at the engine level.
WEBVIEW_OPTION_ENABLE_AUTOFILL	0x00020	Enables form autofill suggestions where available.
WEBVIEW_OPTION_ENABLE_PASSWORD_AUTOSAVE	0x00040	Allows the engine to offer saving website passwords/credentials.
WEBVIEW_OPTION_ENABLE_PRIVACY	0x00080	Enable private browsing
WEBVIEW_OPTION_ENABLE_SCRIPTING	0x00100	Enables JavaScript execution. Disabling blocks script evaluation and script-initiated behaviors.
WEBVIEW_OPTION_ENABLE_DIALOGS	0x00200	Allows script dialogs (alert/confirm/prompt) and related UI. Disable to suppress script-driven dialogs.
WEBVIEW_OPTION_ENABLE_DEVTOOLS	0x00400	Permits opening Developer Tools (e.g., F12). Disable to block DevTools access.
WEBVIEW_OPTION_ENABLE_NEW_WINDOWS	0x00800	Allows the creation of new popup windows
WEBVIEW_OPTION_ENABLE_MESSAGES	0x01000	Enables `window.chrome.webview.postMessage` communication from the page to the host.
WEBVIEW_OPTION_ENABLE_EXTERNAL_DROP	0x02000	Allows external drag-and-drop (e.g., files from Explorer) into the WebView surface.
WEBVIEW_OPTION_RESTRICT_NAVIGATION	0x10000	Blocks navigations after the initial page

		load unless explicitly permitted by the host.
WEBVIEW_OPTION_RESTRICT_DOWNLOADS	0x20000	Blocks attempts to download files through the browser.
WEBVIEW_OPTION_EXCLUSIVE_PROFILE	0x40000	Require exclusive use of the user data folder
WEBVIEW_OPTION_TRANSPARENT_BACKGROUND	0x80000	Renders with a transparent background where content allows; the host window should support composition.
WEBVIEW_OPTION_ENABLE_DEFAULT	0x0031F	Convenience preset enabling the most common interactive features for typical app scenarios.
Navigation Flags		
WEBVIEW_NAVIGATION_REDIRECTED	0x0001	The navigation was the result of an HTTP redirect.
WEBVIEW_NAVIGATION_USER_INITIATED	0x0002	Navigation began from a user gesture rather than being initiated by a script.
WEBVIEW_NAVIGATION_LOCAL_DATA	0x1000	Content was served using local data rather than a full network fetch.
Accelerator Key Flags		
WEBVIEW_ACCELERATOR_KEY_DOWN	0x0001	Indicates an accelerator key-down event.
WEBVIEW_ACCELERATOR_KEY_UP	0x0002	Indicates an accelerator key-up event.
WEBVIEW_ACCELERATOR_KEY_SYSTEM	0x0004	Indicates a system key event using the Alt key, similar to `WM_SYSKEY`.
History Movement Flags		
WEBVIEW_MOVE_BACKWARD_ENABLED	0x0001	Backward navigation is currently available.
WEBVIEW_MOVE_FORWARD_ENABLED	0x0002	Forward navigation is currently available.
Navigation Error Codes		
WEBVIEW_ERROR_NONE	0	No error.
WEBVIEW_ERROR_CERTIFICATE_COMMON_NAME	1	Certificate common name mismatch.
WEBVIEW_ERROR_CERTIFICATE_EXPIRED	2	Certificate is expired.
WEBVIEW_ERROR_INVALID_CLIENT_CERTIFICATE	3	Invalid or untrusted client certificate.
WEBVIEW_ERROR_CERTIFICATE_REVOKED	4	Certificate has been revoked.
WEBVIEW_ERROR_INVALID_CERTIFICATE	5	Certificate is invalid for another reason.
WEBVIEW_ERROR_SERVER_UNREACHABLE	6	Cannot reach the server/host.
WEBVIEW_ERROR_TIMEOUT	7	Operation timed out.
WEBVIEW_ERROR_INVALID_SERVER_RESPONSE	8	Server returned an invalid or unexpected response.

WEBVIEW_ERROR_CONNECTION_ABORTED	9	Connection aborted by the server or network.
WEBVIEW_ERROR_CONNECTION_RESET	10	Connection was reset.
WEBVIEW_ERROR_DISCONNECTED	11	Disconnected during the operation.
WEBVIEW_ERROR_CANNOT_CONNECT	12	Failed to establish a connection.
WEBVIEW_ERROR_HOST_NAME_NOT_RESOLVED	13	DNS resolution failed for the host.
WEBVIEW_ERROR_OPERATION_CANCELED	14	Operation was canceled by the user or host.
WEBVIEW_ERROR_REDIRECT_FAILED	15	Redirection failed.
WEBVIEW_ERROR_UNEXPECTED_ERROR	16	An unexpected error occurred.
WEBVIEW_ERROR_CREDENTIALS_REQUIRED	17	Authentication credentials are required.
WEBVIEW_ERROR_PROXY_AUTHENTICATION_REQUIRED	18	Proxy authentication is required.
Events		
WEBVIEW_EVENT_WINDOW_ATTACHED	1	WebView attached to the host window. This is the first event that fires after the browser is created.
WEBVIEW_EVENT_WINDOW_DETACHED	2	WebView detached from the host window. This is the last event that fires before the browser is destroyed.
WEBVIEW_EVENT_FOCUS_CHANGED	3	WebView focus changed. The ***wParam*** value indicates if focus was gained (1) or lost (0).
WEBVIEW_EVENT_ACCELERATOR_KEYPRESS	4	Accelerator key event. The ***wParam*** value is the virtual-key code and ***lParam*** is the accelerator flags.
WEBVIEW_EVENT_NAVIGATION_STARTING	5	Navigation starting. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the navigation flags.
WEBVIEW_EVENT_NAVIGATION_COMPLETED	6	Navigation completed. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the error code.
WEBVIEW_EVENT_SECURITY_CHANGED	7	Security level changed. The ***wParam*** is the security level and ***lParam*** is the security flags.
WEBVIEW_EVENT_HISTORY_CHANGED	8	Back/forward availability changed. The ***wParam*** value is the history movement flags.
WEBVIEW_EVENT_DOWNLOAD_STARTING	9	A download has been initiated. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the file ID.
WEBVIEW_EVENT_DOWNLOAD_COMPLETED	10	A download has completed. The ***wParam*** value indicates success (1) or failure (0)

		and *lParam* is the file ID.
WEBVIEW_EVENT_MESSAGE_RECEIVED	11	Message received from a script on the page.
WEBVIEW_EVENT_SCRIPT_COMPLETED	12	Script execution completed. The *wParam* value indicates success (1) or failure (0) and *lParam* is the user-defined parameter.
Status Flags		
WEBVIEW_STATUS_SESSION_IDLE	0	No navigation in progress, the browser is in an idle state.
WEBVIEW_STATUS_NAVIGATING	0x0001	Currently navigating to a new location.
WEBVIEW_STATUS_NAVIGATION_ERROR	0x0002	The navigation failed due to an error.
WEBVIEW_STATUS_NAVIGATION_CANCELED	0x0004	The navigation was canceled by the user or host.
WEBVIEW_STATUS_NAVIGATION_TIMEOUT	0x0008	The navigation exceeded the configured timeout period.
WEBVIEW_STATUS_NAVIGATION_BLOCKED	0x0010	The navigation was blocked by policy.
WEBVIEW_STATUS_CONNECTION_FAILED	0x0100	The connection attempt failed or hostname was invalid.
WEBVIEW_STATUS_CONNECTION_ABORTED	0x0200	The connection was aborted or reset.
WEBVIEW_STATUS_SECURITY_WARNING	0x1000	The page includes insecure/mixed content or weak encryption was detected.
Icon Flags		
WEBVIEW_ICON_BIG	0x0001	Update the large window icon.
WEBVIEW_ICON_SMALL	0x0002	Update the small window icon.
WEBVIEW_ICON_BOTH	0x0003	Update both large and small icons.
WEBVIEW_ICON_DPI_AWARE	0x0100	Choose the system-recommended small icon size for the current DPI when possible.
Navigate Flags		
WEBVIEW_NAVIGATE_DEFAULT	0	Default navigation behavior.
WEBVIEW_NAVIGATE_SAFE_URL	0x0001	Applies additional URL safety checks and normalization before navigating.
Header Type Flags		
WEBVIEW_HEADERS_REQUEST	0x0001	Specifies custom request headers supplied by the client.
WEBVIEW_HEADERS_RESPONSE	0x0002	Specifies headers returned by the server in the last response.

Print Dialog Flags		
WEBVIEW_PRINT_DIALOG_DEFAULT	0	Use Edge's print preview dialog, equivalent to the Ctrl+P experience.
WEBVIEW_PRINT_DIALOG_SYSTEM	0x0001	Use the native system print dialog. Does not include print preview.
File Types (Save Document)		
WEBVIEW_FILE_TYPE_DEFAULT	0	Save using a format inferred from the file extension.
WEBVIEW_FILE_TYPE_TEXT	1	Save the document as a plain text file.
WEBVIEW_FILE_TYPE_HTML	2	Save the document as an HTML file.
WEBVIEW_FILE_TYPE_JSON	3	Save the document as JSON formatted string with reserved characters encoded.
WEBVIEW_FILE_TYPE_MHTML	4	Save the document as an MHTML single-file archive which includes all page assets.
WEBVIEW_FILE_TYPE_PDF	5	Save the document as a PDF file.
WEBVIEW_FILE_TYPE_JPEG	101	Save the current view as a JPEG image.
WEBVIEW_FILE_TYPE_PNG	102	Save the current view as a PNG image.
WEBVIEW_FILE_TYPE_WEBP	103	Save the current view as a WEBP image.
Download Status		
WEBVIEW_DOWNLOAD_STARTED	1	The file download has started and is in-progress.
WEBVIEW_DOWNLOAD_COMPLETED	2	The file download has completed successfully.
WEBVIEW_DOWNLOAD_CANCELED	3	The file download has been canceled.
WEBVIEW_DOWNLOAD_FAILED	4	The file download failed due to a network error or other problem.
Zoom		
WEBVIEW_ZOOM_MIN_PERCENT	25	Minimum zoom level as a percentage.
WEBVIEW_ZOOM_MAX_PERCENT	400	Maximum zoom level as a percentage.
WEBVIEW_ZOOM_DEFAULT	100	Default zoom level as a percentage.
Security Levels		
WEBVIEW_SECURITY_LEVEL_UNKNOWN	0	The security state could not be determined.
WEBVIEW_SECURITY_LEVEL_INSECURE	1	The page was retrieved using plain HTTP or the site has an invalid certificate.
WEBVIEW_SECURITY_LEVEL_WARNING	2	The page contains mixed content or only weak encryption is available.
WEBVIEW_SECURITY_LEVEL_NEUTRAL	3	Local content or the browser is in a

		transitional/loading state.
WEBVIEW_SECURITY_LEVEL_SECURE	4	The page was retrieved using HTTPS and the site has a valid certificate.
WEBVIEW_SECURITY_LEVEL_LOW	1	Alias of `WEBVIEW_SECURITY_LEVEL_WARNING`.
WEBVIEW_SECURITY_LEVEL_HIGH	4	Alias of `WEBVIEW_SECURITY_LEVEL_SECURE`.
Security Flags		
WEBVIEW_SECURITY_FLAG_NONE	0	No additional security indicators.
WEBVIEW_SECURITY_FLAG_UNENCRYPTED	0x0001	Content is not encrypted.
WEBVIEW_SECURITY_FLAG_INSECURE	0x0002	Site is insecure using outdated protocols or weak ciphers.
WEBVIEW_SECURITY_FLAG_CERT_WEAK	0x0004	The certificate uses weak signatures.
WEBVIEW_SECURITY_FLAG_CERT_ERROR	0x0008	The certificate is invalid. This is a general error condition.
WEBVIEW_SECURITY_FLAG_CERT_EXPIRED	0x0010	The certificate expired but is otherwise valid.
WEBVIEW_SECURITY_FLAG_UNSAFE	0x0020	The site is considered potentially unsafe (it failed reputation checks).

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WebView Events

The library reports state changes and notifications through a user-defined callback function which is specified when a browser instance is created or attached to a window. For more information, refer to the **WebViewEventProc** function. It is not required for an application to implement handlers for each event. When there is no event handler for a specific event, the browser takes its default action for that event. For example, if your application does not implement a handler for the `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS` event then all accelerator key events are processed normally by the browser control.

Event Constants

Name	Description
WEBVIEW_EVENT_WINDOW_ATTACHED	This event is raised after the WebView2 control has been successfully attached to the specified host window. At this point the browser environment has been created, the controller has been initialized, and the browser instance is ready to accept navigation requests and configuration changes. Applications often use this event to perform initialization tasks such as setting a custom user agent, applying option flags, or starting the first navigation to a default URL. The event confirms that the control is operational and that subsequent API calls will succeed. The *wParam* value indicates success (1) or failure (0). If the browser could not be attached to the window, the *lParam* value will contain the HRESULT code returned by the browser's COM interface.
WEBVIEW_EVENT_WINDOW_DETACHED	This event occurs when the WebView2 control has been detached from its host window, either because the host window is being destroyed or because the application explicitly requested detachment. After this event has fired, the instance is no longer valid and should not be used for navigation or script execution. Applications typically use this event to clean up resources associated with the WebView, save state, or update their user interface to reflect that the browser view is no longer available. Any references to the WebView instance should be considered invalid after detachment.
WEBVIEW_EVENT_FOCUS_CHANGED	This event is raised whenever the browser window gains or loses keyboard focus relative to its host window. The *wParam* value indicates if focus was gained (1) or lost (0). This allows the application to track when user input is directed to the embedded browser versus other controls in the window. For example, you might use this event to update menu state, adjust shortcut key handling, or provide visual cues when the browser window is active.
WEBVIEW_EVENT_ACCELERATOR_KEYPRESS	This event is raised when the user presses an accelerator key combination such as the arrow keys, Page Up and Page Down, Insert, Delete, or other keys that the embedded

	browser normally handles. The ***wParam*** value is the virtual-key code and ***lParam*** is the accelerator flags. The event gives your application a chance to intercept these keystrokes before they are processed by the browser. This is especially useful when you want to provide custom navigation, prevent certain keys from being used, or integrate the control with application-level keyboard shortcuts. To prevent the default behavior being taken by the browser, call the **WebViewKeyEventHandled** function from within this event.
WEBVIEW_EVENT_NAVIGATION_STARTING	This event is fired when a new navigation request begins. The ***wParam*** value indicates success (1) or failure (0) and ***lParam*** is the navigation flags. The event is raised before any network request is made, ensuring that no data is sent until the handler has had a chance to approve or deny the navigation. The **WebViewGetNavigationUrl** function can be called to determine the new URL the browser is navigating to and the **WebViewCancelNavigation** function can be used to prevent the navigation.
WEBVIEW_EVENT_NAVIGATION_COMPLETED	This event is raised when a navigation request has finished loading. It is triggered regardless of whether the navigation succeeded or failed, with the ***wParam*** value indicating success (1) or failure (0). If the navigation has failed, the ***lParam*** value is the error code. Applications typically handle this event to update their UI once the content has loaded, or to implement custom error handling when navigation fails. Because the event always fires, even on errors, it is a reliable notification point for the end of a navigation sequence.
WEBVIEW_EVENT_SECURITY_CHANGED	This event occurs when the security level of the current connection changes. The ***wParam*** value is the security level and ***lParam*** is the security flags. For example, the level may change from neutral to secure once a certificate has been validated, or from secure to warning if mixed content is detected. Additional information about the state of the connection can be obtained by calling the **WebViewGetSecurityInfo** function. Applications often use this event to update visual indicators such as a padlock icon or to apply additional security-related policies. It is important to note that this event may be raised multiple times during a single navigation, and earlier values should be discarded in favor of the most recent state.
WEBVIEW_EVENT_HISTORY_CHANGED	This event occurs whenever the browsing history changes. The ***wParam*** value is the history movement flags. Navigating to a new page, using the **WebViewGoBack** or **WebViewGoForward** functions, or programmatically calling navigation functions can trigger this event. Your application can respond by enabling or disabling UI elements such as toolbar buttons depending on whether it is possible to go

	back or forward.
WEBVIEW_EVENT_DOWNLOAD_STARTING	This event occurs when a file download is initiated either by a user clicking on a link or navigating to a URL which specifies content that cannot be rendered in the browser. The *wParam* value indicates success (1) or failure (0) and *lParam* is the unique file identifier which can be used to identify the specific download. This identifier can be used with the **WebViewGetDownloadInfo** function to obtain information about the download in progress.
WEBVIEW_EVENT_DOWNLOAD_COMPLETED	This event occurs when a file download has completed. The *wParam* value indicates success (1) or failure (0) and *lParam* is the unique file identifier. If the file download is canceled, it is considered to be a failed download. The **WebViewGetDownloadInfo** function can be used to obtain information about the download, such as the total number of bytes copied, how much time elapsed and the full path of the local file that was created.
WEBVIEW_EVENT_MESSAGE_RECEIVED	This event occurs when a message is posted from JavaScript code in the web page using `window.chrome.webview.postMessage`. The message is delivered as a string, which can be retrieved using the **WebViewGetMessage** function. The message may include structured data such as JSON or it may be a simple JSON formatted string. This event provides a two-way communication channel between the page content and the host application, allowing developers to implement hybrid scenarios where the web page can notify the native host of user actions, requests, or state changes. If messages are not enabled, this event will not be raised.
WEBVIEW_EVENT_SCRIPT_COMPLETED	This event is raised when a script execution initiated by the application completes. The *wParam* value indicates success (1) or failure (0) and *lParam* is the user-defined parameter provided when the **WebViewAsyncExecute** function was called. The **WebViewGetResult** function can be called to obtain the result of the script execution. If the script contained errors or could not be executed, the status will reflect a failure condition. Applications typically use this event to retrieve values from the page, execute custom logic in response to script results, or detect when asynchronous script operations have finished. Because script execution is asynchronous, this event is the definitive way to know when a script call has finished.

Remarks

Some events, such as `WEBVIEW_EVENT_NAVIGATION_STARTING` and `WEBVIEW_EVENT_ACCELERATOR_KEYPRESS`, allow your application to influence behavior by canceling an action or handling a keypress. Others, like `WEBVIEW_EVENT_HISTORY_CHANGED` or `WEBVIEW_EVENT_SECURITY_CHANGED`, are informational and allow you to update user interface

elements or record state.

If your application does not enable specific options (such as allowing scripts, dialogs, or message passing), the corresponding events will not be raised. For example, if messages are disabled, the `WEBVIEW_EVENT_MESSAGE_RECEIVED` event will never occur. This makes it important to configure options appropriately for the scenarios you wish to support.

See Also

WebViewAttachWindow, WebViewAttachWindowEx, WebViewCreateWindow, WebViewCreateWindowEx, WebViewEventProc

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Whois Protocol Library

Request registration information for an Internet domain name.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSWHOV12.DLL
Version	12.0.3024.1286
LibID	A3CFB881-EB11-4A10-95A2-C357BF955F74
Import Library	CSWHOV12.LIB
Dependencies	None
Standards	RFC 954

Overview

The Whois protocol library provides an interface for requesting registration information for an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The library provides an API for requesting that information and returning it to the program so that it can be displayed or processed. This library would be most commonly used to query the Whois server at **whois.internic.net** to obtain information about a specific Internet domain name or an administrative contact at that domain.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing

applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

Whois Protocol Functions

Function	Description
WhoisAsyncConnect	Connect asynchronously to the specified server
WhoisAttachThread	Attach the specified client handle to another thread
WhoisCancel	Cancel the current blocking operation
WhoisConnect	Connect to the specified server
WhoisDisableEvents	Disable asynchronous event notification
WhoisDisableTrace	Disable logging of network function calls
WhoisDisconnect	Disconnect from the current server
WhoisEnableEvents	Enable asynchronous event notification
WhoisEnableTrace	Enable logging of network function calls to a file
WhoisEventProc	Callback function that processes events generated by the client
WhoisFreezeEvents	Suspend or resume event handling by the calling process
WhoisGetErrorString	Return a description for the specified error code
WhoisGetLastError	Return the last error code
WhoisGetStatus	Return the current client status.
WhoisGetTimeout	Return the number of seconds until an operation times out
WhoisInitialize	Initialize the library and validates the specified license key at runtime
WhoisIsBlocking	Determine if the client is blocked, waiting for information
WhoisIsConnected	Determine if the client is connected to the server
WhoisIsReadable	Determine if there is data available to be read from the server
WhoisRead	Read data returned by the server
WhoisRegisterEvent	Register an event callback function
WhoisSearch	Search for the specified record
WhoisSetLastError	Set the last error code
WhoisSetTimeout	Set the number of seconds until an operation times out
WhoisUninitialize	Terminate use of the library by the application

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisAsyncConnect Function

```
HCLIENT WINAPI WhoisAsyncConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwReserved,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **WhoisAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **WhoisConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this parameter is NULL, then a synchronous (blocking) connection will be established with the server.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the ***hEventWnd*** parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **WhoisGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has

occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **WhoisDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WhoisConnect, WhoisDisconnect, WhoisInitialize, WhoisSearch

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisAttachThread Function

```
DWORD WINAPI WhoisAttachThread(
    HCLIENT hClient
    DWORD dwThreadId
);
```

The **WhoisAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **WhoisAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **WhoisAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **WhoisCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the ***dwThreadId*** parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **WhoisUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisConnect, WhoisDisconnect, WhoisUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisCancel Function

```
INT WINAPI WhoisCancel(
   HCLIENT hClient
);
```

The **WhoisCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When the **WhoisCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisIsBlocking

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisConnect Function

```
HCLIENT WINAPI WhoisConnect(
   LPCTSTR lpszRemoteHost,
   UINT nRemotePort,
   UINT nTimeout,
   DWORD dwReserved
);
```

The **WhoisConnect** function is used to establish a connection with the specified server.

Parameters

lpszRemoteHost

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **WhoisConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WhoisDisconnect, WhoisInitialize, WhoisSearch

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisDisableEvents Function

```
INT WINAPI WhoisDisableEvents(
    HCLIENT hClient
);
```

The **WhoisDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisEnableEvents, WhoisFreezeEvents, WhoisRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisDisableTrace Function

```
BOOL WINAPI WhoisDisableTrace();
```

The **WhoisDisableTrace** function disables the logging of network function calls file.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisDisconnect Function

```
INT WINAPI WhoisDisconnect(
   HCLIENT hClient
);
```

The **WhoisDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisConnect, WhoisUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisEnableEvents Function

```
INT WINAPI WhoisEnableEvents(
    HCLIENT hClient,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **WhoisEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **WhoisRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the client handle. One or more of the following event identifiers may be sent:

Value	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a

	connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **WhoisDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisDisableEvents, WhoisFreezeEvents, WhoisRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisEnableTrace Function

```
BOOL WINAPI WhoisEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **WhoisEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WhoisDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisEventProc Function

```
VOID CALLBACK WhoisEventProc(
    HCLIENT hClient,
    UINT nEventId,
    DWORD dwError,
    DWORD_PTR dwParam
);
```

The **WhoisEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **WhoisRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **WhoisRegisterEvent** function. The **WhoisEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisDisableEvents, WhoisEnableEvents, WhoisFreezeEvents, WhoisRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisFreezeEvents Function

```
INT WINAPI WhoisFreezeEvents(
    HCLIENT hClient,
    BOOL bFreeze
);
```

The **WhoisFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisDisableEvents, WhoisEnableEvents, WhoisRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisGetErrorString Function

```
INT WINAPI WhoisGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **WhoisGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **WhoisGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WhoisGetLastError, WhoisSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisGetLastError Function

```
DWORD WINAPI WhoisGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **WhoisSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **WhoisGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **WhoisSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those functions which call **WhoisSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisGetErrorString, WhoisSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisGetStatus Function

```
INT WINAPI WhoisGetStatus(
   HCLIENT hClient
);
```

The **WhoisGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

The **WhoisGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Description
WHOIS_STATUS_IDLE	The client is current idle and not sending or receiving data.
WHOIS_STATUS_CONNECT	The client is establishing a connection with the server.
WHOIS_STATUS_READ	The client is reading data from the server.
WHOIS_STATUS_WRITE	The client is writing data to the server.
WHOIS_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisIsBlocking, WhoisIsConnected, WhoisIsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisGetTimeout Function

```
INT WINAPI WhoisGetTimeout(
   HCLIENT hClient
);
```

The **WhoisGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisInitialize Function

```
BOOL WINAPI WhoisInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **WhoisInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WhoisGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **WhoisInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **WhoisUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **WhoisInitialize** or **WhoisUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WhoisConnect, WhoisDisconnect, WhoisUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisIsBlocking Function

```
BOOL WINAPI WhoisIsBlocking(
    HCLIENT hClient
);
```

The **WhoisIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns TRUE. If the client is not performing a blocking operation, or the client handle is invalid, the function returns FALSE.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **WhoisIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisCancel, WhoisGetStatus, WhoisIsConnected, WhoisIsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisIsConnected Function

```
BOOL WINAPI WhoisIsConnected(
    HCLIENT hClient
);
```

The **WhoisIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisGetStatus, WhoisIsBlocking, WhoisIsReadable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisIsReadable Function

```
BOOL WINAPI WhoisIsReadable(
   HCLIENT hClient,
   INT nTimeout,
   LPDWORD lpdwAvail
);
```

The **WhoisIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisGetStatus, WhoisIsBlocking, WhoisIsConnected

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisRead Function

```
INT WINAPI WhoisRead(
   HCLIENT hClient,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **WhoisRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When **WhoisRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisIsBlocking, WhoisIsReadable, WhoisSearch

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisRegisterEvent Function

```
INT WINAPI WhoisRegisterEvent(
   HCLIENT hClient,
   UINT nEventId,
   WHOISEVENTPROC lpfnEvent,
   DWORD_PTR dwParam
);
```

The **WhoisRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Value	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WhoisEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86

(32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **WhoisRegisterEvent** function associates a callback function with a specific event. The event handler is an **WhoisEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisDisableEvents, WhoisEnableEvents, WhoisEventProc, WhoisFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisSearch Function

```
INT WINAPI WhoisSearch(
   HCLIENT hClient,
   LPCTSTR lpszKeyword,
   UINT nSearchType
);
```

The **WhoisSearch** function submits the specified keyword to the server.

Parameters

hClient

Handle to the client session.

lpszKeyword

Points to a string which specifies the query keyword. It may be a handle, name or mailbox.

nSearchType

The type of search being performed. One of the following values may be used:

Value	Description
WHOIS_SEARCH_ANY	Search for any record that matches the specified keyword.
WHOIS_SEARCH_HANDLE	Search only for handles that match the specified keyword.
WHOIS_SEARCH_MAILBOX	Search only for mailboxes that match the specified keyword.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

WhoisConnect, WhoisIsReadable, WhoisRead

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisSetLastError Function

```
VOID WINAPI WhoisSetLastError(
    DWORD dwErrorCode
);
```

The **WhoisSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those functions which call **WhoisSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **WhoisGetLastError** function. The use of **WhoisGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisGetErrorString, WhoisGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisSetTimeout Function

```
INT WINAPI WhoisSetTimeout(
   HCLIENT hClient,
   UINT nTimeout
);
```

The **WhoisSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisGetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

WhoisUninitialize Function

```
VOID WINAPI WhoisUninitialize();
```

The **WhoisUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **WhoisInitialize** call before it can call any of the other the library functions. When it has finished using the library, the application must call **WhoisUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **WhoisUninitialize** for every successful call to **WhoisInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Import Library: cswhov12.lib

See Also

WhoisDisconnect, WhoisInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Windows Sockets Library

A general purpose TCP/IP networking library for developing client and server applications.

Reference

- Functions
- Data Structures
- Error Codes

Library Information

File Name	CSWSKV12.DLL
Version	12.0.3024.1286
LibID	99EEE601-E935-4266-B046-771A823DB47D
Import Library	CSWSKV12.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

At the core of all of the SocketTools networking libraries is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench library provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that developers face when building Internet applications.

This class library supports secure connections using the TLS 1.2 and TLS 1.3 protocols and can also be used to create secure, customized server applications. Both implicit and explicit TLS connections are supported, enabling the class to work with a wide variety of client and server applications without requiring third-party encryption libraries.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on

ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Library Functions

Function	Description
InetAbort	Abort the connection and immediately close the socket
InetAccept	Accept a connection request from a remote host
InetAcceptEx	Accept a client connection on a listening socket with additional options
InetAsyncAccept	Accept an asynchronous connection request from a remote host
InetAsyncAcceptEx	Accept a non-blocking connection on a listening socket
InetAsyncConnect	Connect asynchronously to the specified server
InetAsyncConnectEx	Connect asynchronously to the specified server
InetAsyncListen	Listen for client connections on an asynchronous socket
InetAsyncListenEx	Listen for client connections on an asynchronous socket with additional options
InetAttachSocket	Attach the socket handle to the specified process
InetAttachThread	Attach the specified socket to another thread
InetCancel	Cancel a blocking operation
InetClientBroadcast	Write data to all other clients that are connected to the same server
InetCompareAddress	Compare two IP addresses to determine if they are identical
InetConnect	Connect to the specified server
InetConnectEx	Connect to the specified server with additional connection information
InetConnectUrl	Connect to the specified server using a URL
InetCreateSecurityCredentials	Create a new security credentials structure
InetDeleteSecurityCredentials	Delete a previously created security credentials structure
InetDetachSocket	Detach the socket handle from the current process
InetDisableEvents	Disable asynchronous event notification
InetDisableSecurity	Disable secure communication with the remote host
InetDisableTrace	Disable logging of network function calls
InetDisconnect	Disconnect from the current server
InetEnableEvents	Enable asynchronous event notification
InetEnableSecurity	Enable secure communication with the remote host
InetEnableTrace	Enable logging of network function calls to a file
InetEnumHostAliases	Return a byte array that contains all the aliases for a specified host
InetEnumNetworkAddresses	Return the list of network addresses that are configured for the local host
InetEnumServerClients	Returns a list of active client sessions established with the specified server
InetEnumServerClientsByAddress	Returns a list of active client sessions that match the specified IP address
InetEventProc	Callback method that processes events generated on the socket
InetFindClientMoniker	Returns a handle to the client socket which matches the specified moniker
InetFlush	Flush the send and receive buffers
InetFormatAddress	Convert an IP address in binary format into a printable string
InetFreezeEvents	Suspend or resume event handling by the application
InetGetAdapterAddress	Return the IP or MAC assigned to the specified network adapter

InetGetAddress	Convert an IP address string to a binary format
InetGetAddressFamily	Return the address family for the specified IP address
InetGetBlockingSocket	Return the handle for the socket which is blocked in the current thread
InetGetDefaultHostFile	Return the fully qualified path name of the host file on the local system
InetGetClientData	Returns the application defined data associated with the specified client session
InetGetClientHandle	Returns the handle for a specific client session based on its ID number
InetGetClientId	Returns the unique ID number assigned to the specified client session
InetGetClientIdleTime	Returns the amount of time the specified client session has been idle
InetGetClientMoniker	Returns the string alias associated with the specified client session
InetGetClientPriority	Returns the current priority for the specified client session
InetGetClientServer	Returns a socket handle to the server for the specified client socket
InetGetClientServerById	Returns a socket handle to the server for the specified session identifier
InetGetClientThreadId	Returns the thread ID for the specified client session
InetGetClientThreads	Returns the number of client session threads created by the server
InetGetErrorString	Return a description for the specified error code
InetGetExternalAddress	Return the external IP address assigned to the local system
InetGetHostAddress	Return the IP address assigned to the specified hostname
InetGetHostFile	Return the name of the host file
InetGetHostName	Return the hostname assigned to the specified IP address
InetGetLastError	Return the last error code
InetGetLocalAddress	Return the local IP address and port number for a socket
InetGetLocalName	Return the hostname assigned to the local system
InetGetLockedServer	Return the handle to the server which has been locked
InetGetMemoryUsage	Return the amount of memory allocated for the network connection
InetGetOption	Return the current socket options
InetGetPeerAddress	Return the IP address of the peer that the socket is connected to
InetGetPhysicalAddress	Return the media access control (MAC) address for the primary network adapter
InetGetSecurityInformation	Return information about the security characteristics of a connection
InetGetServerClient	Return the handle for the last client connection accepted by the server
InetGetServerData	Returns the application defined data associated with the specified server
InetGetServerPriority	Return the current priority assigned to the specified server
InetGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
InetGetServerStatus	Returns the status of the specified server
InetGetServerThreadId	Returns the thread ID for the specified server
InetGetServiceName	Return the service name associated with a specified port number
InetGetServicePort	Return the port number associated with a service name
InetGetStatus	Report what sort of socket operation is in progress
InetGetStreamInfo	Return information about the current stream read or write operation
InetGetThreadClient	Return the handle for the client session that is being managed by the specified thread
InetGetTimeout	Return the timeout interval for blocking operations, in seconds

InetGetUrlHostName	Return the host name and port number specified in a URL
InetGetVpnAdapter	Return the name of the active Virtual Private Network adapter
InetHostNameToUnicode	Converts the canonical form of a host name to its Unicode version
InetInitialize	Initialize the library and validate the specified user license key at runtime
InetIsAddressNull	Determine if the specified IP address is a null address
InetIsAddressRoutable	Determine if the specified IP address is routable over the Internet
InetIsBlocking	Determine if the socket is performing a blocking operation
InetIsClosed	Determine if the remote host has closed its socket
InetIsConnected	Determine if the socket is connected to a remote host
InetIsListening	Determine if the socket is listening for a connection
InetIsProtocolAvailable	Determine if the specified protocol and address family are supported
InetIsReadable	Determine if date can be read from the remote process
InetIsUrgent	Determine if there is any out-of-band (OOB) data available to be read
InetIsWritable	Determine if data can be written to the remote process
InetListen	Listen for client connections on the specified socket
InetListenEx	Listen for client connections on the specified socket with additional options
InetMatchHostName	Match a host name against of list of addresses including wildcards
InetNormalizeHostName	Return the canonical form of a host name
InetPeek	Read data from the socket without removing it from the socket buffer
InetRead	Read data from the socket
InetReadEx	Read data from the socket, with extended functionality
InetReadLine	Read a line of data from the socket, storing it in a string buffer
InetReadStream	Read a stream of data from the socket
InetRegisterEvent	Register an event callback function
InetReject	Reject a pending client connection request
InetServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
InetServerBroadcast	Write data to all active clients currently connected to the specified server
InetServerLock	Lock the specified server, causing all other client threads to block until it is unlocked
InetServerRestart	Restart the server, terminating all active client sessions
InetServerResume	Resume accepting client connections on the specified server
InetServerStart	Begin listening for client connections on the specified address and port
InetServerStop	Stop listening for connections and terminate all client sessions
InetServerStopEx	Stop listening for connections and wait for the server to terminate
InetServerSuspend	Suspend accepting client connections on the specified server
InetServerSuspendEx	Suspend accepting client connections and optionally reject or disconnect clients
InetServerThrottle	Limit the number of active client connections, connections per address and connection rate
InetServerUnlock	Unlock the specified server, allowing other client threads to resume execution
InetSetClientData	Associate application defined data with the specified client session
InetSetClientMoniker	Associate a unique string alias with the specified client session

InetSetClientPriority	Set the priority for the specified client session
InetSetServerData	Associate application defined data with the specified server
InetSetServerPriority	Change the priority assigned to the specified server
InetSetServerStackSize	Change the initial size of the stack allocated for threads created by the server
InetSetHostFile	Specify the name of an alternate host table
InetSetLastError	Set the last error code
InetSetOption	Set one or more options for the current socket
InetSetTimeout	Set the interval used when waiting for a blocking operation to complete
InetShutdown	Disable reception or transmission of data
InetStoreStream	Read a stream of data from the remote host and store it in a file
InetUninitialize	Terminate use of the library by the application
InetValidateCertificate	Validate the specified security certificate is installed on the local system
InetValidateHostName	Validate the specified host name and return the resolved IP address
InetWrite	Write data to the socket
InetWriteEx	Write data to the socket, with extended functionality
InetWriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
InetWriteStream	Write a stream of data to the socket

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAbort Function

```
INT WINAPI InetAbort(
   SOCKET hSocket
);
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is 0. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetAbort** function should only be used when the connection must be closed immediately before the application terminates. This function should only be used to abort client connections and should not be used with passive (listening) sockets. Server applications that need to abort an incoming client connection should use the **InetReject** function.

In most cases, the application should call the **InetDisconnect** function to gracefully close the connection to the remote host. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetCancel, InetReject, InetDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAccept Function

```
SOCKET WINAPI InetAccept(
   SOCKET hSocket,
   UINT nTimeout
);
```

The **InetAccept** function is used to accept a pending client connection.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for the connection to complete before failing the operation.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAccept** should be used to exchange information with the client.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and accept the connection by calling **InetAccept** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

To accept a secure connection, use the **InetAcceptEx** function and specify the INET_OPTION_SECURE option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAcceptEx, InetConnect, InetListen, InetRegisterEvent, InetReject, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAcceptEx Function

```
SOCKET WINAPI InetAcceptEx(
   SOCKET hSocket,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **InetAcceptEx** function is used to accept a pending client connection.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the socket.

nTimeout

The number of seconds that the server will wait for a client connection before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has stopped the server.
INET_OPTION_SECURE	This option determines if a secure connection is negotiated with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will

	allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This may be NULL, unless ***dwOptions*** includes INET_OPTION_SECURE. When INET_OPTION_SECURE is used, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAcceptEx** should be used to exchange information with the client.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and accept the connection by calling **InetAcceptEx** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAccept, InetConnect, InetConnectEx, InetListen, InetListenEx, InetRegisterEvent, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAsyncAccept Function

```
SOCKET WINAPI InetAsyncAccept(
    SOCKET hSocket,
    UINT nTimeout,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **InetAsyncAccept** function is used to accept a pending client connection.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for the connection to complete before failing the operation. This value is used only if ***hWnd*** is NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAsyncAccept** should be used to exchange information with the client.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of

	the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAsyncConnect, InetAsyncListen, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAsyncAcceptEx Function

```
SOCKET WINAPI InetAsyncAcceptEx(
   SOCKET hSocket,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **InetAsyncAcceptEx** function is used to accept a pending client connection.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for a client connection before failing the operation. This value is only used with blocking connections.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has stopped

	the server.
INET_OPTION_SECURE	This option determines if a secure connection is negotiated with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This may be NULL, unless dwOptions includes INET_OPTION_SECURE. When INET_OPTION_SECURE is used, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAsyncAccept** or **InetAsyncAcceptEx** should be used to exchange information with the client.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.

INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAsyncAccept, InetAsyncListenEx, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAsyncConnect Function

```
SOCKET WINAPI InetAsyncConnect(
   LPCTSTR lpszHostName,
   UINT nPort,
   UINT nProtocol,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **InetAsyncConnect** function is used to establish a connection with a server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **InetConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Parameters

lpszHostName

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be

	enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the ***lpCredentials*** parameter. If the ***lpCredentials*** parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.

INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When this function is called with UDP as the specified protocol, it does not actually establish a connection. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.

INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAsyncConnectEx, InetConnect, InetDisableEvents, InetDisconnect, InetEnableEvents, InetInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAsyncConnectEx Function

```
SOCKET WINAPI InetAsyncConnectEx(
   LPCTSTR lpszHostName,
   UINT nPort,
   UINT nProtocol,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort,
   LPSECURITYCREDENTIALS lpCredentials,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **InetAsyncConnectEx** function is used to establish a connection with a server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **InetConnectEx** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Parameters

lpszHostName

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the ***lpCredentials*** parameter. If the ***lpCredentials*** parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will

	allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpszLocalAddress

A null-terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the *lParam* parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAsyncConnect, InetConnect, InetDisableEvents, InetDisconnect, InetEnableEvents, InetInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAsyncListen Function

```
SOCKET WINAPI InetAsyncListen(
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **InetAsyncListen** function creates an listening socket.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The socket option INET_OPTION_REUSEADDRESS is enabled by default when calling the **InetAsyncListen** function. This allows an application to re-use a local address and port number when

creating the listening socket. If this behavior is not desired, use the **InetAsyncListenEx** function instead.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_ACCEPT	The process has received a connection request from a client and should accept the connection using the **InetAsyncAccept** function. This event is only generated for server applications which have created an asynchronous socket using the **InetAsyncListen** function.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAsyncAccept, InetAsyncListenEx, InetDisableEvents, InetEnableEvents, InetInitialize, InetListen, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAsyncListenEx Function

```
SOCKET WINAPI InetAsyncListenEx(
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort,
   UINT nBacklog,
   DWORD dwOptions,
   HWND hEventWnd,
   UINT uEventMsg
);
```

The **InetAsyncListenEx** function creates an listening socket.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and

	immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to the address provided by the caller, this function will fail.
INET_OPTION_RESERVEDPORT	This option specifies the listening socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges and firewall rules that will permit a client to establish a connection with the service.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has stopped the server.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use.

If the INET_OPTION_EXCLUSIVE option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also

prevent your server application from restarting immediately, regardless if the INET_OPTION_REUSEADDRESS option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_ACCEPT	An incoming client connection is pending. The connection will be assigned to a new socket. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAsyncAccept, InetAsyncAcceptEx, InetAsyncListen, InetDisableEvents, InetEnableEvents, InetListenEx, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAttachSocket Function

```
SOCKET WINAPI InetAttachSocket(
    SOCKET hSocket
    DWORD dwProcessId
);
```

The **InetAttachThread** function attaches the specified socket handle to another thread.

Parameters

hSocket

Handle to the socket.

dwProcessId

The process ID for the process that currently owns the socket handle. This value may be zero to specify the current process.

Return Value

If the function succeeds, the return value is the handle to the attached socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetAttachSocket** function enables an application to attach an external socket handle to the current thread and initializes it for use by the library. An external socket is any socket handle created directly by the Windows Sockets API or by a third-party library. If the ***dwProcessId*** parameter is zero or specifies the current process ID, then the function checks to see if the socket was created by this library. If it was, then its owner context is switched to the current thread; if the socket was created externally, then it is initialized for use by the library and attached to the current thread.

If the ***dwProcessId*** parameter specifies another process, the socket will be duplicated into the current process, attached to the current thread and the original socket handle will be closed in the other process. This enables an application to effectively take control of a connection created by another process. The original socket handle must be inheritable by the by the current process and must be an actual Windows socket handle, not a pseudo-handle. This functionality is only supported on Windows NT 4.0 and later versions of the operating system with the Microsoft TCP/IP stack. Note that Layered Service Providers (LSPs) may interfere with the ability to inherit handles across processes.

The attached socket is initialized in a blocking state, even if was originally using asynchronous socket events. If the application requires that the socket use events, it must explicitly call **InetEnableEvents** using the handle returned by this function.

In most cases, the handle returned by this function will be the same value as the ***hSocket*** parameter, however an application should never make the assumption that this will be the case. If **InetAttachSocket** returns a socket handle that has a different value than the ***hSocket*** parameter, this indicates that the original handle has been destroyed and should never be used in subsequent function calls.

This function should never be used with a secure socket connection because the attached socket will not have the security context required to encrypt and decrypt the data exchanged with the remote host.

Example

To demonstrate how to pass sockets between processes, this example will use two programs; one

acting as a server to listen for client connections and accept them, the other inheriting the client socket and echoing back anything the client sends. The first program will create the listening socket, and when a client connects, it will call **CreateProcess** to create a new child process to handle that connection.

```
SOCKET hServer;
SOCKET hClient;

// Initialize the library

if (!InetInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    return;

// Listen for incoming client connections

if ((hServer = InetListen(NULL, nLocalPort)) == INVALID_SOCKET)
    return;

while (TRUE)
{
    hClient = InetAccept(hServer, 10);

    if (hClient == INVALID_SOCKET)
    {
        // If InetAccept has timed-out, then simply loop back and attempt
        // to continue accepting connections; otherwise, exit the loop

        if (InetGetLastError() != ST_ERROR_OPERATION_TIMEOUT)
            break;
    }
    else
    {
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
        CHAR szCommandLine[512];
        BOOL bResult;

        // Detach the socket, which will free the memory that the library
        // has allocated for it without actually destroying the socket handle;
        // the child process will close the handle in this process when it
        // attaches to it
        InetDetachSocket(hClient, 0);

        // Initialize the STARTUPINFO structure
        ZeroMemory(&si, sizeof(si));

        // Create the command line arguments, passing the current
        // process ID and socket handle to the new process

        wsprintf(szCommandLine, "%s %lu %lu",
                 lpszAppName,
                 (DWORD)GetCurrentProcessId(),
                 (DWORD)hClient);

        // Create the child process

        bResult = CreateProcess(NULL,
                                szCommandLine,
                                NULL, NULL,
```

```
                                    TRUE,
                                    CREATE_DEFAULT_ERROR_MODE,
                                    NULL, NULL,
                                    &si, &pi);

        if (!bResult)
            InetDisconnect(hClient);
    }
}

InetDisconnect(hServer);
InetUninitialize();
```

The second program attaches to the socket handle that was passed to it by the parent process. It goes into a loop, reading any data sent to it by the client and sending the same data back. When the client disconnects, the **InetRead** function will return 0, it will exit the loop and the process will terminate.

```
SOCKET hSocket;
SOCKET hClient;
DWORD dwProcessId;

// Initialize the library

if (!InetInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    return;

// Process command line arguments that were passed to us
// by the server process

dwProcessId = (DWORD)atol(argv[1]);
hClient = (SOCKET)atol(argv[2]);

// Attach to the hClient socket that the server passed
// to us; this will close the socket in the server process

hSocket = InetAttachSocket(hClient, dwProcessId);

if (hSocket != INVALID_SOCKET)
{
    BYTE cBuffer[512];
    int nRead;

    do
    {
        // Read any data sent to us by the client
        nRead = InetRead(hSocket, cBuffer, sizeof(cBuffer));

        // Echo the data we have read back to the client
        if (nRead > 0)
            InetWrite(hSocket, cBuffer, nRead);
    }
    while (nRead > 0);

    InetDisconnect(hSocket);
}

InetUninitialize();
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAttachThread, InetDetachSocket, InetInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetAttachThread Function

```
DWORD WINAPI InetAttachThread(
    SOCKET hSocket
    DWORD dwThreadId
);
```

The **InetAttachThread** function attaches the specified socket handle to another thread.

Parameters

hSocket

Handle to the socket.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the socket handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When a socket handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in an application to create the socket, and then pass that handle to another worker thread. The **InetAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **InetAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **InetCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **InetUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAccept, InetAcceptEx, InetConnect, InetConnectEx, InetDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetCancel Function

```
INT WINAPI InetCancel(
   SOCKET hSocket
);
```

The **InetCancel** function cancels a blocking operation.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When the **InetCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This function is typically called from within an event handler to signal that the current blocking operation should stop. It may also be used to cancel a blocking operation that is occurring on another thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAbort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetClientBroadcast Function

```
INT WINAPI InetClientBroadcast(
    SOCKET hClient,
    LPBYTE lpBuffer,
    INT cbBuffer
);
```

The **InetClientBroadcast** function sends data to all other clients that are connected to the same server.

Parameters

hClient

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of clients that the data was sent to. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetClientBroadcast** function sends the contents of the buffer to all other clients that are connected to the same server as the specified client. This function will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

This function can only be used with client sessions created as part of the server interface and cannot be used with standard sockets created using the **InetConnect** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetServerBroadcast, InetWrite, InetWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetCompareAddress Function

```
BOOL WINAPI InetCompareAddress(
   LPINTERNET_ADDRESS lpAddress1,
   LPINTERNET_ADDRESS lpAddress2
);
```

The **InetCompareAddress** function compares two Internet addresses in a binary format.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared.

Return Value

If the function succeeds and the two addresses are identical, the return value is non-zero. If the function fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetHostAddress, InetGetLocalAddress, InetGetPeerAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetConnect Function

```
SOCKET WINAPI InetConnect(
   LPCTSTR lpszHostName,
   UINT nRemotePort,
   UINT nProtocol,
   UINT nTimeout,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **InetConnect** function is used to establish a connection with a server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the system you want to connect with. This parameter cannot be a URL and must only specify the name of the remote host. If this parameter is NULL or an empty string, the function will fail with an error indicating the host name is invalid.

nRemotePort

The port number used to establish the connection. Valid port numbers range in value from 1 through 65535 and a value outside if this range will cause the function to fail. Port numbers in the range of 49152 and 65535 are referred to as dynamic ports and are generally reserved for private use by client applications. You cannot specify a port number of zero when establishing an outbound connection.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be

	used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the ***lpCredentials*** parameter. If the ***lpCredentials*** parameter is NULL, the connection will default to using TLS 1.2 or later and the strongest cipher suites available.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6

	address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The ***lpszHostName*** parameter must specify a valid host name or IP address. Host names are resolved into an IP address by first checking the local hosts file and if the name is not found, a name server query will be performed to determine the IP address. If the Unicode version of this function is called and the host name includes non-ASCII characters, the host name will be automatically converted to an ASCII compatible format. Refer to the **InetNormalizeHostName** function for more information. To establish a connection using a URL rather than a host name, use the **InetConnectUrl** function.

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **InetEnableSecurity** function when you are ready to initiate the TLS handshake.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected

behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

When this function is called with UDP as the specified protocol, it does not actually establish a connection in the same way that a TCP stream connection is created. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnectEx, InetConnectUrl, InetDisconnect, InetEnableSecurity, InetInitialize, InetRead, InetRegisterEvent, InetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetConnectEx Function

```
SOCKET WINAPI InetConnectEx(
   LPCTSTR lpszHostName,
   UINT nRemotePort,
   UINT nProtocol,
   UINT nTimeout,
   DWORD dwOptions,
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **InetConnectEx** function is used to establish a connection with a server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the system you want to connect with. This parameter cannot be a URL and must only specify the name of the remote host. If this parameter is NULL or an empty string, the function will fail with an error indicating the host name is invalid.

nRemotePort

The port number used to establish the connection. Valid port numbers range in value from 1 through 65535 and a value outside if this range will cause the function to fail. Port numbers in the range of 49152 and 65535 are referred to as dynamic ports and are generally reserved for private use by client applications. You cannot specify a port number of zero when establishing an outbound connection.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for the connection to complete before failing the operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for

	stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the ***lpCredentials*** parameter. If the ***lpCredentials*** parameter is NULL, the connection will default to using TLS 1.2 or later and the strongest cipher suites available.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a

	connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpszLocalAddress

A null-terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

lpCredentials

A pointer to a SECURITYCREDENTIALS structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The ***lpszHostName*** parameter must specify a valid host name or IP address. Host names are resolved into an IP address by first checking the local hosts file and if the name is not found, a name server query will be performed to determine the IP address. If the Unicode version of this function is called and the host name includes non-ASCII characters, the host name will be automatically converted to an ASCII compatible format. Refer to the **InetNormalizeHostName** function for more information. To establish a connection using a URL rather than a host name, use the **InetConnectUrl** function.

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **InetEnableSecurity** function when you are ready to initiate the TLS handshake.

The ***dwOptions*** argument can be used to specify the threading model that is used by the library when

a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

When this function is called with UDP as the specified protocol, it does not actually establish a connection in the same way that a TCP stream connection is created. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnect, InetConnectUrl, InetDisconnect, InetEnableSecurity, InetInitialize, InetRead, InetRegisterEvent, InetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetConnectUrl Function

```
SOCKET WINAPI InetConnectUrl(
    LPCTSTR lpszUrl,
    UINT nTimeout,
    DWORD dwOptions
);
```

The **InetConnectUrl** function is used to establish a TCP connection with a server using the information provided in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL used when establishing the connection. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the function will fail. See the remarks below for more information on the format supported by this function.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation. If this value is zero, a default timeout period will be used.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is not necessary for most connections, particularly when the client will not be connected to the server for an extended period of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects secure

	connections using the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The connection will always default to using TLS 1.2 or later and the strongest cipher suites available on the client platform. This option may be automatically enabled if the URL scheme specifies a service which requires a secure connection. See the remarks below for more information.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetConnectUrl** function provides a simplified interface which can be used to establish a connection using a URL. This function can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters
...]
```

This function recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. The host name **www.example.com** would be resolved into an IP

address and the connection established on port 443. This function will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the function cannot determine what port number should be used when establishing the connection. The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this function will automatically enable the INET_OPTION_SECURE option. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection regardless if the *dwOptions* parameter includes that option. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

```
http://www.example.com:443
```

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if INET_OPTION_SECURE has not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, this function will return an error indicating the port number is invalid. For example, the URL `https://[2001:db8:0:0:1::128]/` uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this function will only be used to establish a connection with a server. This is a general purpose function which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools API provides comprehensive collection of higher-level functions which can be used to access those services.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **InetEnableSecurity** function when you are ready to initiate the TLS handshake.

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnectUrl** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnect, InetDisconnect, InetEnableSecurity, InetGetUrlHostName, InetInitialize, InetRead, InetRegisterEvent, InetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetCreateSecurityCredentials Function

```
BOOL WINAPI InetCreateSecurityCredentials(
  DWORD dwProtocol,
  DWORD dwOptions,
  LPCTSTR lpszUserName,
  LPCTSTR lpszPassword,
  LPCTSTR lpszCertStore,
  LPCTSTR lpszCertName,
  LPVOID lpvReserved,
  LPSECURITYCREDENTIALS *lppCredentials
);
```

The **InetCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **InetDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **InetGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **InetConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpszPassword* parameter or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
InetCreateSecurityCredentials(SECURITY_PROTOCOL_DEFAULT,
                              CREDENTIAL_STORE_CURRENT_USER,
                              NULL,
                              NULL,
                              strCertStore,
                              strCertName,
                              NULL,
                              &lpSecCred);

hAcceptSocket = InetAsyncAcceptEx(hListenSocket,
                                  nTimeout,
                                  dwOptions,
                                  lpSecCred,
                                  hEventWnd,
                                  uEventMsg);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAcceptEx, InetAsyncAcceptEx, InetAsyncConnectEx, InetConnectEx, InetDeleteSecurityCredentials, InetValidateCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetDeleteSecurityCredentials Function

```
VOID WINAPI InetDeleteSecurityCredentials(
   LPSECURITYCREDENTIALS *lppCredentials
);
```

The **InetDeleteSecurityCredentials** function deletes an existing SECURITYCREDENTIALS structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)
    InetDeleteSecurityCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been established.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetCreateSecurityCredentials, InetValidateCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetDetachSocket Function

```
BOOL WINAPI InetDetachSocket(
    SOCKET hSocket
    DWORD dwThreadId
);
```

The **InetDetachSocket** function detaches the specified socket from the current process.

Parameters

hSocket

Handle to the socket.

dwThreadId

The ID of the thread that owns the socket handle. A value of zero specifies that the current thread is the owner.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetDetachSocket** function will release the memory that the library has allocated for the socket without destroying the socket handle. After the function returns, the socket can no longer be used with other functions in the library, however the socket handle remains valid. This is typically used when passing a socket handle between processes, where the parent process detaches the socket prior to creating the child process. The child then calls **InetAttachSocket** to attach the socket handle to its own process.

This function should never be used with a secure socket connection because detaching a secure socket will force the security context for that session to be released. If the socket is attached to another process, it will not have the security context originally created when the connection was established and will be unable to encrypt or decrypt the data stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAttachThread, InetAttachSocket, InetDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetDisableEvents Function

```
INT WINAPI InetDisableEvents(
    SOCKET hSocket
);
```

The **InetDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetEnableEvents, InetFreezeEvents, InetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetDisableSecurity Function

```
INT WINAPI InetDisableSecurity(
    SOCKET hSocket,
    DWORD dwReserved
);
```

The **InetDisableSecurity** function disables a secure session with the remote host.

Parameters

hSocket

The socket handle.

dwReserved

Reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetDisableSecurity** function disables a secure session, with subsequent calls to **InetRead** and **InetWrite** sending and receiving unencrypted data. It is important to note that because this function sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This function does not close the socket. Use the **InetDisconnect** function to close the socket and release the resources allocated for the current session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetCreateSecurityCredentials, InetDeleteSecurityCredentials, InetEnableSecurity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetDisableTrace Function

```
BOOL WINAPI InetDisableTrace();
```

The **InetDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetEnableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetDisconnect Function

```
INT WINAPI InetDisconnect(
    SOCKET hSocket
);
```

The **InetDisconnect** function terminates the connection, closing the socket and releasing the memory allocated for the session.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

Once the connection has been terminated, the socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

After a socket is closed, it will go into a TIME-WAIT state which prevents an application from using the same source and destination address and port numbers bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. This can have an impact on an application that rapidly connects and disconnects over a short period of time because it can exhaust the pool of ephemeral ports.

If this function is called using a server socket handle returned by the **InetServerStart** function, all active client connections will be disconnected, the listening socket will be closed and the server thread will terminate. If this function is called with a client socket handle allocated by the server, it will terminate the client connection and the thread that manages it.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetConnect, InetConnectEx, InetServerStart, InetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnableEvents Function

```
INT WINAPI InetEnableEvents(
    SOCKET hSocket,
    HWND hEventWnd,
    UINT nEventMsg
);
```

The **InetEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **InetRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hSocket

The socket handle.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

nEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_ACCEPT	The process has received a connection request from a client and should accept the connection using the **InetAsyncAccept** function. This event is only generated for server applications which have created an asynchronous socket using the **InetAsyncListen** function.
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the ***lParam*** parameter should be checked, since this notification message will also be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional

	messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The process has timed-out waiting for a blocking operation to complete. This event is only generated for synchronous sockets.
INET_EVENT_CANCEL	The application has canceled a blocking operation. This event is fired once an operation has been terminated by the **InetCancel** function, and control has been returned to the calling process.

This function cannot be used with sockets that are created by the SocketWrench server interface. Those sockets are managed separately in their own thread, and event notifications are handled inside the callback function specified when the server is created using the **InetServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetDisableEvents, InetFreezeEvents, InetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnableSecurity Function

```
INT WINAPI InetEnableSecurity(
   SOCKET hSocket,
   DWORD dwOptions,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **InetEnableSecurity** function enables a secure session with the remote host.

Parameters

hSocket

The socket handle.

dwOptions

An unsigned integer value that specifies additional security options. It may have a value of zero or one of the following options:

Value	Description
INET_SECURE_CLIENT	The certificate specified by the ***lpCredentials*** parameter will be used as a client certificate, and the application will begin to negotiate the secure session as a client by initiating the handshake with the server. The certificate that is used must be a valid client certificate with a private key associated with it. If the ***lpCredentials*** parameter is NULL, then a secure client session will be initiated without a client certificate.
INET_SECURE_SERVER	The certificate specified by the ***lpCredentials*** parameter will be used as a server certificate, and the application will wait for the remote host to initiate the handshake that establishes the parameters of the secure session. The certificate that is used must be a valid server certificate and have a private key associated with it. The ***lpCredentials*** parameter cannot be NULL if this option is specified.

lpCredentials

Pointer to a SECURITYCREDENTIALS structure. This parameter may be NULL, in which case no client credentials will be provided. If client credentials are required, the structure members ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the SECURITYCREDENTIALS structure.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetEnableSecurity** function enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate. This function is useful if the application needs to establish a standard connection to the remote host and then negotiate a secure connection at a later point (this is known as explicit TLS). If the function succeeds, all subsequent calls to **InetRead** and **InetWrite** to receive and send data will be encrypted.

If the *dwOptions* parameter has a value of zero and the socket was created using **InetConnect** or related functions to establish a client connection, then **InetEnableSecurity** will initiate the handshake with the remote host to establish a secure session. If the **InetAccept** or related functions were used to accept a connection from a client, then the function will block and wait for the client to initiate the handshake.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnect, InetConnectEx, InetCreateSecurityCredentials, InetDeleteSecurityCredentials, InetDisableSecurity

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnableTrace Function

```
BOOL WINAPI InetEnableTrace(
   LPCTSTR lpszTraceFile,
   DWORD dwTraceFlags
);
```

The **InetEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

	multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "**0x**". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxxx:xxxx::xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetDisableTrace

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnumHostAliases Function

```
INT WINAPI InetEnumHostAliases(
   LPCTSTR lpszHostName,
   INT nAddressFamily,
   LPTSTR lpszHostAliases,
   INT nMaxLength
);
```

The **InetEnumHostAliases** function returns a collection of null-terminated strings which contain the aliases for a specified host.

Parameters

lpszHostName

A null-terminated string which specifies a host name or IP address. This parameter cannot be NULL and must specify a valid host name.

nAddressFamily

An integer which specifies the address family which should be used when resolving the host name or IP address. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	If the host name specifies an IP address, it will be resolved based on the format of the string. For compatibility, host names are resolved to IPv4 addresses by default but if there is only an IPv6 address assigned to the host name, it will be used.
INET_ADDRESS_IPV4	Specifies the host name should be resolved using an IPv4 address.
INET_ADDRESS_IPV6	Specifies the host name should be resolved using an IPv6 address.

lpszHostAliases

A null-terminated string buffer which will contain the aliases for the specified. If this parameter is NULL, the function will return the number of characters which would be copied into the buffer.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer, including the terminating null characters. If the ***lpszHostAliases*** parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is the number of characters copied to the string, including the terminating null characters which indicate the end of each host alias. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If there are multiple aliases for the host name, each name will be terminated with a null character, with an extra null character indicating the end of the list of aliases.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnumNetworkAddresses Function

```
INT WINAPI InetEnumNetworkAddresses(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddressList,
   INT nMaxAddresses
);
```

The **InetEnumNetworkAddresses** function returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses assigned to the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that require support for IPv6 connections.
INET_ADDRESS_IPV4	Return only the IPv4 addresses assigned to the local host. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Return only the IPv6 addresses assigned to the local host. All bytes in the ***ipNumber*** array are significant. This option is only recommended for those applications that require support for IPv6 connections.

lpAddressList

A pointer to an array of INTERNET_ADDRESS structures that will contain the IP address of each local network interface. This parameter may be NULL, in which case the method will only return the number of available addresses.

nMaxAddresses

Maximum number of addresses to be returned. If the ***lpAddressList*** parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the number of network addresses that are configured for the local host. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_ANY, the application must be prepared to accept IPv6 addresses returned by this function. On Windows Vista and later versions of the operating system, IPv6 support is enabled and the local network adapter will have IPv6 addresses assigned to them by default. For legacy applications that only recognize IPv4 addresses, the

nAddressFamily parameter should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

This function will ignore addresses that are bound to a disabled interface, as well as those addresses bound to a virtual loopback interface. For example, although the loopback address 127.0.0.1 is a valid network address, it will not be included in list of addresses returned by this function.

The first IPv4 or IPv6 address returned by this function is typically the address assigned to the primary network adapter on the local system. However, your application should not depend on addresses being returned in any particular order. If the system has dial-up networking or virtualization software installed, this function may also include the IP addresses assigned to any virtualized network adapters installed by that software.

Example

```
INTERNET_ADDRESS *lpAddressList = NULL;
INT nAddressCount = InetEnumNetworkAddresses(INET_ADDRESS_IPV4, NULL, 0);

if (nAddressCount > 0)
{
    // Allocate memory for the array of IP addresses
    lpAddressList = (INTERNET_ADDRESS *)LocalAlloc(LPTR, nAddressCount *
sizeof(INTERNET_ADDRESS));

    if (lpAddressList == NULL)
    {
        // Virtual memory exhausted
        return;
    }

    // Populate the array with the addresses
    nAddressCount = InetEnumNetworkAddresses(INET_ADDRESS_IPV4, lpAddressList,
nAddressCount);
}

_tprintf(_T("There are %d local network addresses assigned\n"), nAddressCount);

// Display each IP address assigned to the local system
for (INT nIndex = 0; nIndex < nAddressCount; nIndex++)
{
    TCHAR szValue[64];

    // Convert the IP address to a printable string
    InetFormatAddress(lpAddressList + nIndex, szValue, 64);
    _tprintf(_T("%d: %s\n"), nIndex, szValue);
}

// Free the memory allocated for the IP address list
if (lpAddressList != NULL)
    LocalFree((HLOCAL)lpAddressList);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetAdapterAddress, InetGetHostAddress, InetGetLocalAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnumServerClients Function

```
INT WINAPI InetEnumServerClients(
    SOCKET hServer,
    SOCKET *lpClients,
    INT nMaxClients
);
```

The **InetEnumServerClients** function returns a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

lpClients

Pointer to an array of socket handles which identifies all client connections. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client socket handles to be returned. If the ***lpClients*** parameter is NULL, this parameter should be specified with a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. A return value of zero indicates that there are no active client sessions. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections, the function will fail. To dynamically determine the number of active connections, call the function with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero. To enumerate the active clients that match a specific IP address, use the **InetEnumServerClientsByAddress** function.

This function will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this function will return zero, indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server. To determine the actual number of client sessions regardless of their status, use the **InetGetClientThreads** function.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Example

```
INT nMaxClients = InetEnumServerClients(hServer, NULL, 0);

if (nMaxClients > 0)
{
    SOCKET *lpClients = NULL;

    // Allocate memory for client sockets
```

```
    lpClients = (SOCKET *)LocalAlloc(LPTR, nMaxClients * sizeof(SOCKET));
    if (lpClients == NULL)
    {
        // Virtual memory has been exhausted
        return;
    }

    nMaxClients = InetEnumServerClients(hServer, lpClients, nMaxClients);
    if (nMaxClients == INET_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nClient = 0; nClient < nMaxClients; nClient++)
    {
        // Perform some action with each client socket
        SOCKET hClient = lpClients[nClient];
    }

    // Free memory allocated for client sockets
    LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetEnumServerClientsByAddress, InetGetClientThreads, InetServerBroadcast, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEnumServerClientsByAddress Function

```
INT WINAPI InetEnumServerClientsByAddress(
   SOCKET hServer,
   LPINTERNET_ADDRESS lpAddress,
   SOCKET *lpClients,
   INT nMaxClients
);
```

The **InetEnumServerClientsByAddress** function returns a list of active client sessions established with the specified server that match the specified IP address.

Parameters

hServer

Handle to the server socket.

lpAddress

Pointer to an INTERNET_ADDRESS structure that specifies the client IP address. If this parameter is NULL, then all active client sessions will be enumerated.

lpClients

Pointer to an array of socket handles which identifies all client connections that match the specified IP address. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client socket handles to be returned. If the ***lpClients*** parameter is NULL, this parameter should be specified with a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server that match the specified IP address. A return value of zero indicates that no clients have connected from the specified IP address. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the ***nMaxClients*** parameter is less than the number of active client connections that match the specified address, the function will fail. To dynamically determine the number of active connections, call the function with the ***lpClients*** parameter with a value of NULL, and the ***nMaxClients*** parameter with a value of zero. If the ***lpAddress*** parameter is NULL, this function will return a list of all active clients connected to the server and is effectively the same as calling the **InetEnumServerClients** function. If an address is provided, it must be a valid IP address in a supported address family.

This function will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this function will return zero, indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Example

```
INTERNET_ADDRESS ipAddress;
```

```
if (InetGetPeerAddress(hSocket, &ipAddress, NULL) == INET_ERROR)
{
    // Unable to get the peer IP address for the socket
    return;
}

// Determine the number of clients that have the same IP address
INT nMaxClients = InetEnumServerClientsByAddress(hServer, &ipAddress, NULL, 0);

if (nMaxClients > 0)
{
    SOCKET *lpClients = NULL;

    // Allocate memory for client sockets
    lpClients = (SOCKET *)LocalAlloc(LPTR, nMaxClients * sizeof(SOCKET));
    if (lpClients == NULL)
    {
        // Virtual memory has been exhausted
        return;
    }

    nMaxClients = InetEnumServerClientsByAddress(hServer, &ipAddress, lpClients,
nMaxClients);
    if (nMaxClients == INET_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nClient = 0; nClient < nMaxClients; nClient++)
    {
        // Perform some action with each client socket
        SOCKET hClient = lpClients[nClient];
    }

    // Free memory allocated for client sockets
    LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetEnumServerClients, InetGetClientThreads, InetServerBroadcast, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetEventProc Function

```
VOID CALLBACK InetEventProc(
   SOCKET hSocket,
   UINT nEvent,
   DWORD dwError,
   DWORD_PTR dwParam
);
```

The **InetEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hSocket

The socket handle.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the InetRegisterEvent function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **InetRegisterEvent** function. The **InetEventProc** function is a placeholder for the application-defined function name.

If the callback function is being used with the **InetServerStart** function, the function will be called in the context of the thread that is currently managing the server or client session. You must ensure that any access to global or static variables is synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself. If you need to manage state information for a specific client, then use the **InetGetClientData** and **InetSetClientData** functions which will allow you to access application defined data for that client session in a thread-safe manner.

When this callback function is used for event notifications from the server interface, the *hSocket* parameter specifies the client socket handle, except for the INET_EVENT_ACCEPT event, in which case the handle references the server socket handle. To obtain the handle of the client connection that was just accepted, use the **InetGetServerClient** function. To obtain the handle to the server using the client socket handle, use the **InetGetClientServer** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetDisableEvents, InetEnableEvents, InetFreezeEvents, InetGetClientServer, InetGetServerClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetFindClientMoniker Function

```
SOCKET WINAPI InetFindClientMoniker(
    SOCKET hServer,
    LPCTSTR lpszMoniker
);
```

The **InetFindClientMoniker** function returns a handle to the client socket which matches the specified moniker.

Parameters

hServer

A handle to the server.

lpszMoniker

A null-terminated string which specifies the client moniker to search for. This parameter cannot be NULL and cannot specify an empty string.

Return Value

If the function succeeds, the return value is the handle to the client socket for the session that matches the specified moniker. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **InetSetClientMoniker** function. This function will search all active client sessions for the server, and returns the socket handle to the client that matches the specified moniker. If there is no match, an error will be returned.

The moniker can be any string value, however monikers are not case sensitive and may not contain embedded null characters. The maximum length of a moniker is 127 characters.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetClientMoniker, InetGetClientServer, InetGetClientThreadId, InetSetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetFlush Function

```
INT WINAPI InetFlush(
   SOCKET hSocket
);
```

The **InetFlush** function flushes the internal send and receive buffers used by the socket.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetFlush** function will flush any data waiting to be read or written to the remote host . It is important to note that this function is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling **InetWrite** or prior to calling **InetDisconnect**.

An application never needs to use **InetFlush** under normal circumstances. This function is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsReadable, InetIsWritable, InetPeek, InetRead, InetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetFormatAddress Function

```
INT WINAPI InetFormatAddress(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszAddress,
   INT cchAddress
);
```

The **InetFormatAddress** function converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

A null-terminated string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the function succeeds, the return value is the length of the IP address string. If the function fails, the return value is INET_ERROR, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the INTERNET_ADDRESS structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetExternalAddress, InetGetHostAddress, InetGetLocalAddress, InetGetPeerAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetFreezeEvents Function

```
INT WINAPI InetFreezeEvents(
    SOCKET hSocket,
    BOOL bFreeze
);
```

The **InetFreezeEvents** function is used to suspend and resume event handling.

Parameters

hSocket

Socket handle.

bFreeze

A non-zero value specifies that event handling should be suspended. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if event handling has been suspended, and four read events occur, only one of those read events will be posted to the application when even handling is resumed. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetDisableEvents, InetEnableEvents, InetRegisterEvent

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetAdapterAddress Function

```
INT WINAPI InetGetAdapterAddress(
    INT nAdapterIndex,
    INT nAddressType,
    LPTSTR lpszAddress,
    INT nMaxLength
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Value	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the function fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetAdapterAddress** function will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned

dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This function returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this function would return zero.

This function will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this function may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    TCHAR szAddress[64];
    INT cchAddress;

    cchAddress = InetGetAdapterAddress(nIndex, INET_ADAPTER_IPV4, szAddress, 64);

    if (cchAddress == INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetEnumNetworkAddresses, InetGetLocalAddress, InetGetLocalName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetAddress Function

```
INT WINAPI InetGetAddress(
   LPCTSTR lpszAddress,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetGetAddress** function converts an IP address string to binary format.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the ***lpszAddress*** parameter. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the ***lpszAddress*** parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the ***lpszAddress*** parameter does not specify a valid IPv4 address string, this function will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the ***ipNumber*** array are significant. If the ***lpszAddress*** parameter does not specify a valid IPv6 address string, this function will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system,

use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetFormatAddress, InetIsAddressNull, InetIsAddressRoutable, InetIsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetAddressFamily Function

```
INT WINAPI InetGetAddressFamily(
   LPCTSTR lpszAddress,
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the function succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the function fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call **InetGetLastError**.

Value	Description
INET_ADDRESS_IPV4	The address passed to the function is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the function is a valid IPv6 address.

Remarks

The **InetGetAddressFamily** function returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other functions such as **InetConnect**. Note that this function will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetIsAddressNull, InetIsAddressRoutable, InetIsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetBlockingSocket Function

```
SOCKET InetGetBlockingSocket();
```

The **InetGetBlockingSocket** function returns the handle for the socket in the current thread which is currently blocking, if there is one.

Parameters

None.

Return Value

If the function succeeds, the return value is the handle for the socket in the current thread which is currently blocking. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAbort, InetCancel

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientData Function

```
BOOL WINAPI InetGetClientData(
    SOCKET hClient,
    LPVOID *lppvData
);
```

The **InetGetClientData** function returns the application defined data associated with the specified client session.

Parameters

hClient

The client socket handle.

lppvData

Pointer to a void pointer which will contain an application defined value associated with the client session.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be retrieved. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientData** function is used to retrieve the application defined data that was previously associated with a client session using the **InetSetClientData** function. This is typically used to associate a pointer to a data structure or a class instance with a specific client handle.

This function can only be used with client socket handles created using the server interface. It cannot be used with socket handles created using the **InetConnect** or **InetAccept** functions. If the socket handle is invalid, or does not reference a client socket handle created by the server, the *lppvData* pointer passed to this function will be initialized to a value of NULL and the function will return a value of zero.

If this function is called with a valid socket handle and there is no data associated with the socket, the function will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this function, the application should always check the return value to ensure the function succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));
UINT *pnValue2 = NULL;

*pnValue1 = 1234;

if (InetSetClientData(hClient, pnValue1) == FALSE)
{
    // Unable to associate the data with this session
    return;
}

if (InetGetClientData(hClient, &pnValue2) == FALSE)
{
```

```
    // Unable to retrieve the data associated with this session
    return;
}

// *pnValue2 == 1234
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerData, InetSetClientData, InetSetServerData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientHandle Function

```
SOCKET WINAPI InetGetClientHandle(
    SOCKET hServer,
    UINT nClientId
);
```

The **InetGetClientHandle** function returns the handle for a specific client session based on its ID number.

Parameters

hServer

Handle to the server socket.

nClientId

An unsigned integer value which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the socket handle for the specified client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **InetGetClientId** function and used by the application to identify that client session. The **InetGetClientHandle** function can then be used to used to obtain the client socket handle for the session, based on that client ID.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientId, InetGetClientMoniker, InetSetClientMoniker, InetGetServerClient, InetGetThreadClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientId Function

```
UINT WINAPI InetGetClientId(
    SOCKET hClient
);
```

The **InetGetClientId** function returns the unique ID number assigned to the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which uniquely identifies the client session. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **InetGetClientId** function and used by the application to identify that client session. The **InetGetClientHandle** function can then be used to used to obtain the client socket handle for the session, based on that client ID. It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientHandle, InetGetClientMoniker, InetSetClientMoniker, InetGetServerClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientIdleTime Function

```
DWORD WINAPI InetGetClientIdleTime(
     SOCKET hClient
);
```

Returns the number of milliseconds that the specified client session has been idle.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which specifies the number of milliseconds the client session has been idle. If the function fails, the return value is INFINITE. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientIdleTime** function will return the number of milliseconds that have elapsed since data was exchanged with the client. The elapsed time is limited to the resolution of the system timer, which is typically in the range of 10 milliseconds to 16 milliseconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientHandle, InetGetClientMoniker, InetGetServerClient

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientMoniker Function

```
INT WINAPI InetGetClientMoniker(
   SOCKET hSocket,
   LPTSTR lpszMoniker,
   INT nMaxLength
);
```

The **InetGetClientMoniker** function returns the moniker associated with the specified client session.

Parameters

hSocket

Handle to the client socket.

lpszMoniker

Pointer to a string buffer that will contain the moniker for the specified client session when the function returns.

nMaxLength

The maximum number of characters that may be copied into the string buffer. The buffer must be large enough to store the moniker and a terminating null character. The maximum length of a moniker is 127 characters.

Return Value

If the function succeeds, the return value is the number of characters in the moniker string. A return value of zero specifies that no moniker was assigned to the socket. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **InetSetClientMoniker** function. This function will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **InetFindClientMoniker** function.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetFindClientMoniker, InetGetClientHandle, InetGetClientId, InetSetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientPriority Function

```
INT WINAPI InetGetClientPriority(
    SOCKET hClient
);
```

The **InetGetClientPriority** function returns the current priority for the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is the priority for the specified client session. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientPriority** function can be used to determine the current priority assigned to the specified client session. The client priority is inherited from the priority specified when the server is started using the **InetServerStart** function. It may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerPriority, InetServerStart, InetSetClientPriority, InetSetServerPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientServer Function

```
SOCKET WINAPI InetGetClientServer(
    SOCKET hClient
);
```

The **InetGetClientServer** function returns a socket handle to the server for the specified client socket.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientServer** function returns the handle to the server that created the client session. The **InetGetClientServerById** function can be used to obtain the server handle using the client session ID rather than the client socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientHandle, InetGetClientId, InetGetClientServerById

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientServerById Function

```
SOCKET WINAPI InetGetClientServerById(
     UINT nClientId
);
```

The **InetGetClientServerById** function returns a socket handle to the server for the specified client session identifier.

Parameters

nClientId

Client session identifier.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientServerById** function returns the handle to the server that created the client session using the client's unique identifier. The **InetGetClientServer** function can be used to obtain the server handle using the client socket handle rather than the client session ID. This function is typically used in conjunction with the INET_NOTIFY_CONNECT notification message to obtain the handle to the server that generated the event using the client ID passed in the ***wParam*** message parameter.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientHandle, InetGetClientId, InetGetClientServer, InetServerAsyncNotify

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientThreadId Function

```
DWORD WINAPI InetGetClientThreadId(
     SOCKET hClient
);
```

The **InetGetClientThreadId** function returns the thread ID for the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which identifies the thread that was created to manage the client session. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The thread ID returned by this function can be used with the **OpenThread** function to obtain a handle to the thread. Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientPriority, InetGetServerThreadId, InetGetThreadClient, InetSetClientPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetClientThreads Function

```
INT WINAPI InetGetClientThreads(
    SOCKET hServer
);
```

Returns the number of client session threads created by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the number of client session threads that have been created by the server. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientThreads** function returns the number of threads that are managing client sessions for the specified server. If there are no clients connected to the server, this function will return a value of zero. Because this function returns the number of session threads, the value returned will include those clients that are in the process of disconnecting from the server but their session thread has not yet terminated. This differs from the **InetEnumServerClients** function which will only enumerate active clients.

If you wish to determine when the last client has disconnected from the server, call this function within an event handler for the INET_EVENT_DISCONNECT event. If the function returns a value greater than one, then there are other client sessions that are either connected or in the process of terminating.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetEnumServerClients, InetEnumServerClientsByAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetDefaultHostFile Function

```
INT WINAPI InetGetDefaultHostFile(
    LPTSTR lpszFileName,
    INT nMaxLength
);
```

The **InetGetDefaultHostFile** function returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The location of the default host file depends on the operating system. For Windows 95/98 and Windows Me the file is stored in C:\Windows\hosts and for Windows NT and later versions the file is stored in C:\Windows\system32\drivers\etc\hosts. Regardless of platform, there is no filename extension and this file may or may not exist on a given system.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **InetGetLastError**.

Remarks

This function only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **InetSetHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetHostAddress, InetGetHostFile, InetGetHostName, InetSetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetErrorString Function

```
INT WINAPI InetGetErrorString(
   DWORD dwErrorCode,
   LPTSTR lpszDescription,
   INT nMaxLength
);
```

The **InetGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the InetGetLastError function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetLastError, InetSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetExternalAddress Function

```
INT WINAPI InetGetExternalAddress(
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetGetExternalAddress** function returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return any external IP address available. IPv4 addresses will be preferred over IPv6 for backwards compatibilty.
INET_ADDRESS_IPV4	Specifies that the external address should be in IPv4 format.
INET_ADDRESS_IPV6	Specifies that the external address should be in IPv6 format. If the local system does not have IPv6 enabled, the function will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure which contains the external IP address of the local system when the function returns.

Return Value

If the function succeeds, the return value is zero and the INTERNET_ADDRESS structure contains the external IP address for the local host in binary form. If the function fails, it will return INET_ERROR and the *ipFamily* member of the INTERNET_ADDRESS structure will be set to INET_ADDRESS_UNKNOWN. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetExternalAddress** function returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **InetGetLocalAddress** function will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **InetGetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

An application should always check the return value in case there is an error. Never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local system for a number of reasons, particularly if it is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library. If you are using a VPN (Virtual Private Network), this function will return the external IP address for the VPN adapter and not your local network connection. The **InetGetVpnAdapter** function can be used to obtain information about the active VPN adapter.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a background worker thread and call this function from within that thread.

To convert the address from its binary form into a string, use the **InetFormatAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetFormatAddress, InetGetHostAddress, InetGetLocalAddress, InetGetPeerAddress, InetGetVpnAdapter, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetHostAddress Function

```
INT WINAPI InetGetHostAddress(
   LPCTSTR lpszHostName,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetGetHostAddress** function resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

The name of the host to resolve; this may be a fully-qualified domain name or an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the specified host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function can also be used to convert an address in dot notation to a binary format. If the function must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the ***ipFamily*** member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned,

however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpszHostName* parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetHostName, InetGetLocalAddress, InetGetLocalName, InetGetPeerAddress, InetIsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetHostFile Function

```
INT WINAPI InetGetHostFile(
   LPTSTR lpszFileName,
   INT nMaxLength
);
```

The **InetGetHostFile** function returns the name of the host file previously set using the **InetSetHostFile** function. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the function was unable to determine the file name. To get extended error information, call **InetGetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the function failed.

Remarks

This function only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **InetSetHostFile** function.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this function may be different than the default host file for the local system. To determine the file name for the default host file, use the **InetGetDefaultHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetDefaultHostFile, InetGetHostAddress, InetGetHostName, InetSetHostFile

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetHostName Function

```
INT WINAPI InetGetHostName(
   LPINTERNET_ADDRESS lpAddress,
   LPTSTR lpszHostName,
   INT cchHostName
);
```

The **InetGetHostName** function performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the IP address that should be resolved into a host name.

lpszHostName

The buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the function succeeds, the return value is the length of the hostname. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the function must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This function requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this function frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetHostAddress, InetGetLocalAddress, InetGetLocalName, InetGetPeerAddress, InetGetUrlHostName, InetIsProtocolAvailable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetLastError Function

```
DWORD WINAPI InetGetLastError();
```

Parameters

None.

Return Value

Functions set this value by calling the InetSetLastError function. The return value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **InetGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **InetSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those functions which call **InetSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetErrorString, InetSetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetLocalAddress Function

```
INT WINAPI InetGetLocalAddress(
   SOCKET hSocket,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnPort
);
```

The **InetGetLocalAddress** function returns the local IP address and port number for the specified socket.

Parameters

hSocket

The socket handle.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the type of connection that was established. If the ***hSocket*** parameter is INVALID_SOCKET, a preference will be given for returning an IPv4 address. However, if the local host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the local host. If the ***hSocket*** parameter is specified as INVALID_SOCKET, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If the ***hSocket*** parameter specifies a valid socket, this parameter will be set to the local port that the socket was bound to. If the ***hSocket*** parameter is specified as INVALID_SOCKET, this parameter is ignored. If the port number is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the ***ipFamily*** member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the ***nAddressFamily*** parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for the local host to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the ***nAddressFamily*** member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **InetGetLocalAddress** function will return the local, non-routable IP address assigned to the system. To determine the public IP address has been assigned to the system, you should use the **InetGetExternalAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetExternalAddress, InetGetHostAddress, InetGetHostName, InetGetLocalName, InetGetPeerAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetLocalName Function

```
INT WINAPI InetGetLocalName(
   LPTSTR lpszHostName,
   INT cchHostName
);
```

The **InetGetLocalName** function returns the hostname assigned to the local system.

Parameters

lpszHostName

The buffer that will contain the hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the function succeeds, the return value is the length of the hostname. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetHostAddress, InetGetHostName, InetGetLocalAddress, InetGetPeerAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetLockedServer Function

```
SOCKET WINAPI InetGetLockedServer(
    LPDWORD lpdwThreadId
);
```

The **InetGetLockedServer** function unlocks the specified server, allowing other server threads to resume execution.

Parameters

lpdwThreadId

A pointer to an unsigned integer which identifies the thread that established the server lock. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is the handle to the locked server. If no server is in a locked state, the function will return a value of INVALID_SOCKET.

Remarks

The **InetGetLockedServer** function can be used to determine if there is a server in a locked state. If there is, the function will return a handle to the server and will identify the thread which established the lock. This function may be called from any thread, however only the thread which established the server lock may interact with the server or release the lock.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetServerLock, InetServerUnlock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetMemoryUsage Function

```
BOOL WINAPI InetGetMemoryUsage(
   SOCKET hSocket,
   ULARGE_INTEGER *lpMemUsage
);
```

Return the amount of memory allocated for the network connection.

Parameters

hSocket

The socket handle.

lpMemUsage

A pointer to a ULARGE_INTEGER variable which will specify how much memory has been allocated for the socket. This parameter will be initialized to a value of zero by the function and updated with the total number of bytes allocated.

Return Value

If the function succeeds, the return value is non-zero and the memory usage value will be updated. If the socket handle does not specific a socket created by the library, the return value is zero. Call the **InetGetLastError** function to determine the cause of the failure.

Remarks

This function returns the amount of memory allocated for the socket. It enumerates all heap allocations made by the process and returns the total number of bytes allocated for the specific socket handle. If the ***hSocket*** parameter value is INVALID_SOCKET, the function will calculate the amount of memory allocated for all sockets created by the library. This value only reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This function forces the current thread into a locked state, and all network operations will block until the function returns. Because this function enumerates all heaps created for the process, it can be an expensive operation, particularly when a large number of connections have been established. Frequent use of this function can significantly degrade the performance of the application. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active network connections. If the value returned by the function remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStackSize, InetSetServerStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetOption Function

```
INT WINAPI InetGetOption(
   SOCKET hSocket,
   DWORD dwOption,
   LPBOOL lpbEnabled
);
```

The **InetGetOption** function is used to determine if a specific socket option has been enabled.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

lpbEnabled

A pointer to a boolean flag. If the option is enabled, the flag is set to a non-zero value, otherwise it is set to a value of zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAsyncConnectEx, InetConnectEx, InetSetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetPeerAddress Function

```
INT WINAPI InetGetPeerAddress(
   SOCKET hSocket,
   LPINTERNET_ADDRESS lpAddress,
   UINT *lpnPort
);
```

The **InetGetPeerAddress** function returns the peer IP address and remote port number for the specified socket.

Parameters

hSocket

The socket handle.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the remote host that the socket is connected to.

lpnPort

A pointer to an unsigned integer that will contain the remote port that the socket is connected to.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If this function is called by a server application in response to a INET_EVENT_ACCEPT event, it will return the IP address and port number for the client that is attempting to establish the connection. If the peer address is unavailable, the *ipFamily* member of the INTERNET_ADDRESS structure will be zero. To convert the IP address to a printable string, use the **InetFormatAddress** function.

It is not recommended that you use the port number for anything other than informational and logging purposes. Server applications should not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetFormatAddress, InetGetHostAddress, InetGetHostName, InetGetLocalAddress, InetGetLocalName, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetPhysicalAddress Function

```
BOOL WINAPI InetGetPhysicalAddress(
    LPTSTR lpszAddress,
    UINT nMaxLength
);
```

Return the media access control (MAC) address for the primary network adapter.

Parameters

lpszAddress

A string buffer that will contain the address in a printable format when the function returns. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetPhysicalAddress** function returns the media access control (MAC) address for the primary network adapter. This is a 48 bit or 64 bit address that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This function returns the MAC address as a printable string, with each byte of the address as a two-digit hexadecimal value separated by a colon. The string buffer passed to the function should be at least 20 characters long to accommodate the address and terminating null character. An example of a 48 bit address would be "01:23:45:67:89:AB". If the local system is multi-homed (having more than one network adapter) then this function will return the MAC address for the primary network adapter.

This function is provided for backwards compatibility with previous versions of the library and it is recommended that new applications use the **InetGetAdapterAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetEnumNetworkAddresses, InetGetAdapterAddress, InetGetLocalName, InetGetHostAddress

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetSecurityInformation Function

```
BOOL WINAPI InetGetSecurityInformation(
   SOCKET hSocket,
   LPSECURITYINFO lpSecurityInfo
);
```

The **InetGetSecurityInformation** function fills a structure with information about the security characteristics of a connection.

Parameters

hSocket

Handle to the socket.

lpSecurityInfo

A pointer to a SECURITYINFO structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;

securityInfo.dwSize = sizeof(SECURITYINFO);
if (InetGetSecurityInformation(hClient, &securityInfo))
{
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)
    {
        MessageBox(NULL, _T("The connection is not secure"),
                         _T("Connection"), MB_OK);
    }
    else
    {
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)
        {
            MessageBox(NULL, _T("The connection is secure"),
                             _T("Connection"), MB_OK);
        }
        else
        {
            MessageBox(NULL, _T("The server certificate not valid"),
                             _T("Connection"), MB_OK);
        }
```

```
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnectEx, InetAsyncConnectEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServerClient Function

```
SOCKET WINAPI InetGetServerClient(
    SOCKET hServer
);
```

The **InetGetServerClient** function returns the handle for the last client connection accepted by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the handle to the last client connection that was accepted by the server. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerClient** function can be used inside the service event handler to determine the client connection that was just accepted by the server. This would typically be used in conjunction with the INET_EVENT_ACCEPT handler, enabling the application to obtain the handle of the new client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientThreadId, InetGetServerThreadId, InetServerBroadcast, InetServerLock, InetServerStop, InetServerThrottle, InetServerUnlock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServerData Function

```
BOOL WINAPI InetGetServerData(
    SOCKET hServer,
    LPVOID *lppvData
);
```

The **InetGetServerData** function returns the application defined data associated with the specified server.

Parameters

hServer

The server socket handle.

lppvData

Pointer to a void pointer which will contain an application defined value associated with the server.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the server could not be retrieved. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerData** function is used to retrieve the application defined data that was previously associated with a server using the **InetSetServerData** function. This is typically used to associate a pointer to a data structure or a class instance with a specific instance of a server.

This function can only be used with socket handles created using **InetServerStart** function. It cannot be used with socket handles created using the **InetListen** or **InetListenEx** functions. If the socket handle is invalid, or does not reference a server handle, the *lppvData* pointer passed to this function will be initialized to a value of NULL and the function will return a value of zero.

If this function is called with a valid socket handle and there is no data associated with the socket, the function will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this function, the application should always check the return value to ensure the function succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));
UINT *pnValue2 = NULL;

*pnValue1 = 1234;

if (InetSetServerData(hServer, pnValue1) == FALSE)
{
    // Unable to associate the data with this server
    return;
}

if (InetGetServerData(hServer, &pnValue2) == FALSE)
{
    // Unable to retrieve the data associated with this server
    return;
}
```

```
// *pnValue2 == 1234
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientData, InetSetClientData, InetSetServerData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServerPriority Function

```
INT WINAPI InetGetServerPriority(
    SOCKET hServer
);
```

The **InetGetServerPriority** function returns the current priority for the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerPriority** function can be used to determine the current priority assigned to the specified server. It will be one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientPriority, InetServerStart, InetSetClientPriority, InetSetServerPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServerStackSize Function

```
DWORD WINAPI InetGetServerStackSize(
     SOCKET hServer
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetMemoryUsage, InetServerStart, InetSetServerStackSize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServerStatus Function

```
INT WINAPI InetGetServerStatus(
    SOCKET hServer
);
```

The **InetGetServerStatus** function returns the current status of the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the status for the specified server. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerStatus** function can be used to determine the current status for the specified server. It may be one of the following values:

Value	Description
INET_SERVER_INACTIVE	The server is currently inactive.
INET_SERVER_STARTED	The server has initialized and is preparing to listen for client connections.
INET_SERVER_LISTENING	The server is actively listening for incoming client connections.
INET_SERVER_SUSPENDED	The server has been suspended and is no longer accepting client connections.
INET_SERVER_SHUTDOWN	The server has been stopped and is in the process of terminating all active client connections.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetServerRestart, InetServerResume, InetServerStart, InetServerStop, InetServerSuspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServerThreadId Function

```
DWORD WINAPI InetGetServerThreadId(
     SOCKET hServer
);
```

The **InetGetServerThreadId** function returns the thread ID for the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which identifies the thread that was created to manage the server. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The thread ID returned by this function can be used with the **OpenThread** function to obtain a handle to the thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientPriority, InetGetClientThreadId, InetGetThreadClient, InetSetClientPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServiceName Function

```
BOOL WINAPI InetGetServiceName(
   UINT nServicePort,
   LPTSTR lpszServiceName,
   INT nMaxLength
);
```

The **InetGetServiceName** function returns the service name associated with a specified port number.

Parameters

nServicePort

Port number associated with some network service.

lpszServiceName

A null-terminated string buffer that will contain the name of the service associated with the specified port number. This string should be at least 32 characters long.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetServicePort

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetServicePort Function

```
UINT WINAPI InetGetServicePort(
   LPCTSTR lpszServiceName
);
```

The **InetGetServicePort** function returns the port number associated with a service name.

Parameters

lpszServiceName

A null-terminated string which specifies the name of the service to return the port number for.

Return Value

If the function succeeds, the return value is the port number associated with a service name. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetServiceName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetStatus Function

```
INT WINAPI InetGetStatus(
   SOCKET hSocket
);
```

The **InetGetStatus** function is used to report what sort of socket operation is in progress.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The return value is one of the following values:

Value	Description
INET_STATUS_UNUSED	No connection has been established.
INET_STATUS_IDLE	The socket is idle and not in a blocked state
INET_STATUS_LISTEN	The socket is listening for inbound connections from a client
INET_STATUS_CONNECT	The socket is establishing a connection with a server
INET_STATUS_ACCEPT	The socket is accepting a connection from a client
INET_STATUS_READ	Data is being read from the socket
INET_STATUS_WRITE	Data is being written to the socket
INET_STATUS_FLUSH	The socket is being flushed; all data in the receive buffers is being discarded
INET_STATUS_DISCONNECT	The socket is disconnecting from the remote host

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetBlockingSocket, InetIsConnected, InetIsListening, InetIsReadable, InetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetStreamInfo Function

```
BOOL WINAPI InetGetStreamInfo(
   SOCKET hSocket,
   LPINETSTREAMINFO lpStreamInfo
);
```

The **InetGetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

hSocket

Handle to the socket.

lpStreamInfo

A pointer to an INETSTREAMINFO structure which contains information about the status of the network stream.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetStreamInfo** function returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this function will return the status of the last successful streaming read or write performed by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetReadStream, InetStoreStream, InetWriteStream, INETSTREAMINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetThreadClient Function

```
SOCKET WINAPI InetGetThreadClient(
    DWORD dwThreadId
);
```

The **InetGetThreadClient** function returns the socket handle for the client session that is being managed by the specified thread.

Parameters

dwThreadId

An unsigned integer value which identifies the thread managing the client session. If this parameter has a value of zero, then the client handle for the current thread is returned.

Return Value

If the function succeeds, the return value is the socket handle for the specified client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetThreadClient** is used to obtain the socket handle for the client session that is being managed by the specified thread. If the specified thread ID is zero, then the function will return the client socket for the current thread, otherwise it will search the internal table of all active client sessions and return the handle to the session that is being managed by that thread.

This function will fail if the thread ID does not specify an active client session thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientHandle, InetGetClientId, InetGetClientThreadId, InetGetServerClient, InetGetServerThreadId

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetTimeout Function

```
INT WINAPI InetGetTimeout(
    SOCKET hSocket
);
```

The **InetGetTimeout** function returns the timeout interval for blocking operations, in seconds.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is the timeout interval for blocking operations, in seconds. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetSetTimeout

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetUrlHostName Function

```
INT WINAPI InetGetUrlHostName(
   LPCTSTR lpszUrl,
   DWORD dwReserved,
   LPTSTR lpszHostName,
   INT nMaxLength,
   UINT *lpnHostPort,
   UINT *lpnProtocol,
   DWORD *lpdwOptions
);
```

The **InetGetUrlHostName** function returns the host name and port number specified in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the function will fail. See the remarks below for more information on the format supported by this function.

dwReserved

An unsigned integer value reserved for internal use. This value should always be zero.

lpszHostName

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name.

nMaxLength

The maximum number of characters that can be copied to the *lpszHostName* string buffer. This parameter cannot be zero, and must include the terminating null character.

lpnHostPort

Pointer to an unsigned integer value which will contain the port number specified in the URL. This parameter value will always be initialized by the function with a value of zero. If this parameter is NULL, it will be ignored and no port information will be returned.

lpnProtocol

Pointer to an unsigned integer value which will contain the protocol associated with the URI scheme. This parameter value will always be initialized by the function with a value of zero. If this parameter is NULL, it will be ignored and no protocol information will be returned.

lpdwOptions

Pointer to an unsigned integer value which will contain any socket options required to establish a connection based on the URI scheme or specified port. This parameter value will always be initialized by the function with a value of zero. If this parameter is NULL, it will be ignored.

Return Value

If the function succeeds, the return value is the number of characters copied into the *lpszHostName* buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetUrlHostName** function will extract the host name and port number from a URL and return the canonical form of the host name. If the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters have been specified, they will contain the port number, protocol and additional connection options associated with the URL scheme.

The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters
...]
```

This function recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com/
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. This function will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

If the *lpszUrl* parameter only specifies a host name without a URI scheme or port number, this function will ignore the *lpnHostPort, lpnProtocol* and *lpdwOptions* parameters and return the canonical form of the host name in the *lpszHostName* string argument.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, the function will return an error indicating the port number is invalid. For example, the URL `https://[2001:db8:0:0:1::128]/` uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

If the URL uses an IP address instead of a host name, this function will return a copy of that IP address in the *lpszHostName* string provided by the caller. The function will not attempt to resolve the IP address into a host name, however you can use the **InetGetHostName** function to perform a reverse DNS lookup if required.

The only URI schemes currently supported by this function use TCP stream connections. In practical terms, this means the *lpnProtocol* parameter will always return with the value INET_PROTOCOL_TCP when the function is successful. If the function fails, this value will be INET_PROTOCOL_NONE.

Although this function performs checks to ensure that the *lpszUrl* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the host name. To check if the host name exists and has a valid IP address, use the **InetValidateHostName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnectUrl, InetGetHostAddress, InetGetHostName, InetHostNameToUnicode, InetNormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetGetVpnAdapter Function

```
BOOL WINAPI InetGetVpnAdapter(
   LPTSTR lpszAdapterName,
   INT nMaxLength,
   INT nAddressFamily,
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetGetVpnAdapter** function returns the name of the active VPN adapter.

Parameters

lpszAdapterName

A null-terminated string buffer which will contain the name of the active VPN adapter if one is found. If a VPN was not detected, the buffer will contain an empty string when the function returns. This parameter may be NULL if the adapter name is not required.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. If this value is zero, the ***lpszAdapterName*** parameter is ignored.

nAddressFamily

An integer value which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return the IP address for the adapter in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the adapter only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the adapter address should be returned in IPv4 format. The first four bytes of the ***ipNumber*** member of the INTERNET_ADDRESS structure are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the adapter address should be returned in IPv6 format. All bytes in the ***ipNumber*** member of the INTERNET_ADDRESS structure are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address assigned to the VPN adapter when the function returns. This parameter may be NULL if this information is not required. If this parameter is NULL, the value of the ***nAddressFamily*** parameter is ignored.

Return Value

If the function succeeds, the return value is non-zero if an active VPN adapter was found. If no VPN adapter was detected, this function will return zero.

Remarks

The **InetGetVpnAdapter** function provides a best-effort method to detect if the system is currently using a Virtual Private Network (VPN) connection. The detection logic is based on identifying active network adapters with names or descriptions that match known VPN software or common VPN adapter naming conventions, while explicitly excluding adapters associated with virtualization platforms (such as Hyper-V, VMware, and VirtualBox), loopback interfaces, and other non-VPN virtual adapters.

Because there is no industry standard for how VPN clients are identified at the network adapter level, this function uses a heuristic approach and cannot guarantee detection in every scenario. Some VPN clients may use custom adapter names or attempt to obscure their presence, and new VPN software may not be recognized until the internal detection list is updated. There is also a small chance that non-VPN software using similar naming conventions could result in a false positive.

The result of this function should be treated as a hint for diagnostic or troubleshooting purposes. It should not be used to restrict access to your application or deny service to users based solely on the possible presence of a VPN connection. If you specify an address family and the ***lpAddress*** parameter is not NULL, it will contain the first IP address bound to the adapter interface. It is possible for VPN adapters to be bound to multiple IP addresses. The **InetFormatAddress** function can be used to convert the INTERNET_ADDRESS value to a printable string.

The adapter name returned by this function is the description of the detected VPN adapter, encoded as a UTF-8 string in the ANSI version of this function. If you require more detailed network information, consider using the **InetEnumNetworkAddresses** function. You can use the **InetGetExternalAddress** function to obtain the public-facing IP address allocated for the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetEnumNetworkAddresses, InetFormatAddress, InetGetAdapterAddress, InetGetExternalAddress, InetGetLocalAddress, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetHostNameToUnicode Function

```
INT WINAPI InetHostNameToUnicode(
   LPCTSTR lpszHostName,
   LPTSTR lpszUnicodeName,
   INT nMaxLength
);
```

The **InetHostNameToUnicode** function converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the ***lpszUnicodeName*** string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **InetHostNameToUnicode** function will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this function, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the Unicode version of this function is used, the value returned in ***lpszUnicodeName*** will be a Unicode string using UTF-16 encoding. If the ANSI version of this function, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this function performs checks to ensure that the ***lpszHostName*** parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the InetGetHostAddress function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetHostAddress, InetNormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetInitialize Function

```
BOOL WINAPI InetInitialize(
   LPCTSTR lpszLicenseKey,
   LPVOID lpvReserved
);
```

The **InetInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **InetInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **InetUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **InetInitialize** or **InetUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetConnect, InetConnectEx, InetDisconnect, InetListen, InetUninitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsAddressNull Function

```
BOOL WINAPI InetIsAddressNull(
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetIsAddressNull** function determines if the IP address is null.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure that contains the address to check.

Return Value

If the function succeeds and the IP address is null, or the ***lpAddress*** parameter is a NULL pointer, the return value is non-zero. If the function fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetAddress, InetIsAddressRoutable, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsAddressRoutable Function

```
BOOL WINAPI InetIsAddressRoutable(
   LPINTERNET_ADDRESS lpAddress
);
```

The **InetIsAddressRoutable** function determines if the IP address is routable over the Internet.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure that contains the address to check. This parameter cannot be NULL.

Return Value

If the function succeeds and the IP address is routable over the Internet, the return value is non-zero. If the function fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetAddress, InetGetExternalAddress, InetIsAddressNull, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsBlocking Function

```
BOOL WINAPI InetIsBlocking(
    SOCKET hSocket
);
```

The **InetIsBlocking** function is used to determine if the socket is performing a blocking operation.

Parameters

hSocket

Socket handle.

Return Value

If the socket is currently performing a blocking operation, the function returns a non-zero value. If the socket is not performing a blocking operation, or the socket handle is invalid, the function returns zero.

Remarks

This function is typically used to determine if an open socket that is being used by another thread is currently blocked. A socket may block when waiting to receive data from a remote host or while data is actively being exchanged. Because there can only be one blocking socket operation per thread, this function can be used to determine if a function such as **InetRead** or **InetWrite** would fail because another thread is currently sending or receiving data on that socket. This socket handle that is passed to this function does not need to be owned by the current thread.

It is important to note that if this function returns a non-zero value, it does not guarantee that a subsequent read or write on the socket will succeed. The application should always check the return value from functions such as **InetRead** and **InetWrite** to ensure they were successful.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsConnected, InetIsReadable, InetIsWritable, InetRead, InetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsClosed Function

```
BOOL WINAPI InetIsClosed(
   SOCKET hSocket
);
```

The **InetIsClosed** function is used to determine if the remote host has closed its socket.

Parameters

hSocket

Socket handle.

Return Value

If the remote host has closed its socket, the function returns a non-zero value. If the remote host has not closed its connection, or the socket handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsConnected, InetIsListening, InetIsReadable, InetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsConnected Function

```
BOOL WINAPI InetIsConnected(
    SOCKET hSocket
);
```

The **InetIsConnected** function is used to determine if the socket is currently connected to a remote host.

Parameters

hSocket

Socket handle.

Return Value

If the socket is connected to a remote host, the function returns a non-zero value. If the socket is not connected, or the socket handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsClosed, InetIsListening, InetIsReadable, InetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsListening Function

```
BOOL WINAPI InetIsListening(
    SOCKET hSocket
);
```

The **InetIsListening** function determines if the socket is listening for connection requests.

Parameters

hSocket

Socket handle.

Return Value

If the socket is being used to listen for connection requests, the function returns a non-zero value. If the socket is not listening or the socket handle is invalid, the function returns zero.

Remarks

The **InetIsListening** function determines if the socket is being used in a server application to actively listen for incoming connection requests from client applications. A listening socket can be created using either the **InetAsyncListen** or **InetListen** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAsyncListen, InetIsReadable, InetIsWritable, InetIsConnected, InetListen

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsProtocolAvailable Function

```
BOOL WINAPI InetIsProtocolAvailable(
    INT nAddressFamily,
    INT nProtocol
);
```

The **InetIsProtocolAvailable** function determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the system is capable of creating a socket using the specified address family and protocol, this function will return a non-zero value. If the combination of address family and protocol is not supported, this function will return a value of zero.

Remarks

The **InetIsProtocolAvailable** function is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!InetIsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetAddress, InetGetHostAddress, InetGetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsReadable Function

```
BOOL WINAPI InetIsReadable(
   SOCKET hSocket,
   DWORD dwTimeout,
   LPDWORD lpdwAvail
);
```

The **InetIsReadable** function is used to determine if data is available to be read from the socket.

Parameters

hSocket

Socket handle.

dwTimeout

Timeout value in milliseconds. If the socket cannot be read within this time period, the function will return a value of zero. A timeout value of zero specifies that the socket should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read.

Return Value

If the current thread can read data from the socket without blocking, the function returns a non-zero value. If the current thread cannot read any data without blocking, the function returns zero.

Remarks

On some platforms, the value returned in *lpdwAvail* will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsClosed, InetIsWritable, InetPeek, InetRead, InetReadLine, InetReadStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsUrgent Function

```
BOOL WINAPI InetIsUrgent(
   SOCKET hSocket
);
```

The **InetIsUrgent** function determines if there is any out-of-band (OOB) data available to be read.

Parameters

hSocket

Handle to the socket.

Return Value

If there is out-of-band data, the return value is non-zero. If there is no out-of-band data, or an error occurs the return value is zero. To determine if an error has occurred, use the **InetGetLastError** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetSetOption (INET_OPTION_INLINE)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetIsWritable Function

```
BOOL WINAPI InetIsWritable(
    SOCKET hSocket,
    DWORD dwTimeout
);
```

The **InetIsWritable** function is used to determine if data can be written to the socket.

Parameters

hSocket

Socket handle.

dwTimeout

Timeout value in milliseconds. If the socket cannot be written to within this time period, the function will return a value of zero. A timeout value of zero specifies that the socket should be polled without blocking the current thread.

Return Value

If the current thread can write data to the socket within the timeout period, the function returns a non-zero value. The function will return zero if the socket send buffer is full.

Remarks

The **InetIsWritable** function cannot be used to determine the amount of data that can be sent to the remote host without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **InetWriteStream** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsReadable, InetWrite, InetWriteLine, InetWriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetListen Function

```
SOCKET WINAPI InetListen(
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort
);
```

The **InetListen** function creates a passive socket used to listen for connections from a client application.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The socket option INET_OPTION_REUSEADDRESS is enabled by default when calling the **InetListen** function. This allows an application to re-use a local address and port number when creating the listening socket. If this behavior is not desired, use the **InetListenEx** function instead.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

After the listening socket has been created, the application should then call the **InetAccept** function to wait for a client to establish a connection. For servers that need to handle multiple simultaneous client connections, it is recommended that the asynchronous functions be used.

Example

```
SOCKET hServer = INVALID_SOCKET;
LPCTSTR lpszAddress = _T("192.168.0.48");
```

```
// Accept connections from clients that connect to
// address 192.168.0.48 on port 7000

hServer = InetListen(lpszAddress, 7000);
if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];

    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAccept, InetAcceptEx, InetInitialize, InetListenEx, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetListenEx Function

```
SOCKET WINAPI InetListenEx(
   LPCTSTR lpszLocalAddress,
   UINT nLocalPort,
   UINT nBacklog,
   DWORD dwOptions
);
```

The **InetListenEx** function creates a passive socket, and specifies the maximum number of connection requests that will be queued.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is

	for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to the address provided by the caller, this function will fail.
INET_OPTION_RESERVEDPORT	This option specifies the listening socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges and firewall rules that will permit a client to establish a connection with the service.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

If the INET_OPTION_EXCLUSIVE option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the INET_OPTION_REUSEADDRESS option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

After the listening socket has been created, the application should then call the **InetAccept** or **InetAcceptEx** function to wait for a client to establish a connection. For servers that need to handle multiple simultaneous client connections, it is recommended that the asynchronous functions be used.

Example

```
SOCKET hServer = INVALID_SOCKET;
LPCTSTR lpszAddress = _T("192.168.0.48");

// Accept connections from clients that connect to
// address 192.168.0.48 on port 7000 with a standard
// backlog of 5 connections

hServer = InetListenEx(lpszAddress,
                       7000,
                       INET_BACKLOG,
                       INET_OPTION_REUSEADDRESS);

if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];

    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetAccept, InetAcceptex, InetListen, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetMatchHostName Function

```
BOOL WINAPI InetMatchHostName(
   LPCTSTR lpszHostName,
   LPCTSTR lpszHostMask
   BOOL bResolve
);
```

The **InetMatchHostName** function matches a host name against one more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetMatchHostName** function provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the function would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the function would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetAddress, InetGetHostAddress, InetGetHostName, InetGetLocalAddress, InetGetPeerAddress

---

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetNormalizeHostName Function

```
INT WINAPI InetNormalizeHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszNormalized,
   INT nMaxLength
);
```

The **InetNormalizeHostName** function returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name.

nMaxLength

The maximum number of characters that can be copied to the ***lpszNormalized*** string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetNormalizeHostName** function will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this function, it will be converted to an ASCII compatible format for domain names.

The ***lpszHostName*** parameter should only specify a host name or IP address. If you want to support the use of URLs to establish a connection, use the **InetGetUrlHostName** function which has extended support for extracting the host name and port number specified in a URL.

If the Unicode version of this function is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

Although this function performs checks to ensure that the ***lpszHostName*** parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the **InetValidateHostName** function.

It is recommended that you use this function if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this function prior to calling other SocketWrench functions which accept a host name as a parameter. Those functions already normalize the host name and perform checks to ensure it is in the correct format.

If the *lpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the **InetGetHostName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetHostAddress, InetGetHostName, InetGetUrlHostName, InetHostNameToUnicode, InetValidateHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetPeek Function

```
INT WINAPI InetPeek(
   SOCKET hSocket,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **InetPeek** function reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpBuffer

Pointer to the buffer in which the data will be copied. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the ***lpBuffer*** parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetPeek** function returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this function is not removed from the socket buffers. It must be consumed by a subsequent call to the **InetRead** or **InetReadEx** function. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **InetIsReadable** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetFlush, InetRead, InetReadEx, InetWrite, InetWriteEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetRead Function

```
INT WINAPI InetRead(
   SOCKET hSocket,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **InetRead** function reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetRead** function will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this function is called, the current thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This function will return if any amount of data is sent by the remote host, and will not block until the entire buffer has been filled. To avoid blocking the current thread, either create an asynchronous socket or use the **InetIsReadable** function to determine if there is data available to be read prior to calling this function.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the remote host is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** function will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the remote host is writing string data to the socket, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the remote host is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **InetReadLine** function will return a line of text at a time and perform this conversion for you.

When **InetRead** is called and the socket is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is

available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetIsReadable, InetPeek, InetReadEx, InetWrite, InetWriteEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetReadEx Function

```
INT WINAPI InetReadEx(
   SOCKET hSocket,
   LPVOID lpvBuffer,
   INT cbBuffer,
   DWORD dwReserved,
   LPINTERNET_ADDRESS lpRemoteAddress,
   UINT *lpnRemotePort
);
```

The **InetReadEx** function reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpvBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

dwReserved

Reserved parameter. This value must always be zero.

lpRemoteAddress

Pointer to an INTERNET_ADDRESS structure which will contain the IP address of the remote host that sent the data being read. If this information is not required, the parameter may be specified as NULL.

lpnRemotePort

Pointer to an unsigned integer which will contain the remote port number. If this information is not required, the parameter may be specified as NULL.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When **InetReadEx** is called and the socket is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

This function extends the **InetRead** function to return additional information about the peer who sent the data being received. For a client TCP socket, the IP address and remote port are the same values that were used to establish the connection. For a server TCP socket, it is the IP address and port number of the client which sent the data. When reading data from a UDP socket, this is the IP address

and remote port of the peer that sent the datagram. This information can be used in conjunction with the **InetWriteEx** function to send a datagram back to that host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetFlush, InetGetPeerAddress, InetPeek, InetRead, InetWrite, InetWriteEx, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetReadLine Function

```
BOOL WINAPI InetReadLine(
   SOCKET hSocket,
   LPTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **InetReadLine** function reads up to a line of data from the socket and returns it in a string buffer.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReadLine** function reads data from the socket and copies into a specified string buffer. Unlike the **InetRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a null-terminated string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **InetReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **InetReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **InetRead** and **InetReadLine** function calls can be intermixed, however be aware that **InetRead**

will consume any data that has already been buffered by the **InetReadLine** function and this may have unexpected results.

Unlike the **InetRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = InetReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = InetGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetIsReadble, InetRead, InetWrite, InetWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetReadStream Function

```
BOOL WINAPI InetReadStream(
   SOCKET hSocket,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions,
   LPBYTE lpMarker,
   DWORD cbMarker,
   DWORD dwReserved
);
```

The **InetReadStream** function reads the socket data stream and stores the contents in the specified buffer.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpvBuffer

Pointer to the buffer that will contain or reference the data when the function returns. The actual argument depends on the value of the ***dwOptions*** parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the function returns. This argument should should always point to an initialized value. If the lpvBuffer argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the ***lpvBuffer*** parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the function will return a handle which references the data; otherwise, the function will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the function returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The ***lpvBuffer*** argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the function returns. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned

	integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The ***lpvBuffer*** argument specifies a pointer to a global memory handle. When the function returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the function does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this function.
INET_STREAM_HANDLE	The ***lpvBuffer*** argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The ***lpvBuffer*** argument specifies a socket handle. The data read from the socket specified by the ***hSocket*** argument will be written to this socket. The socket handle passed to this function must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either attach the socket using the InetAttachSocket function or use the INET_STREAM_HANDLE stream buffer type instead.
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this function.

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the function, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the function will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReadStream** function enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **InetRead** method, which will return immediately when any amount of data has been read, **InetReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the function returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Because **InetReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
HGLOBAL hgblBuffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400;   // Read up to 100K bytes
BOOL bResult;

bResult = InetReadStream(hSocket,
                         &hgblBuffer,
                         &cbBuffer,
                         INET_STREAM_HGLOBAL | INET_STREAM_CONVERT,
                         NULL, 0, 0);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Use data in the stream buffer
```

```
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetStreamInfo, InetRead, InetReadLine, InetStoreStream, InetWrite, InetWriteLine, InetWriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetRegisterEvent Function

```
INT WINAPI InetRegisterEvent(
   SOCKET hSocket,
   UINT nEventId,
   INETEVENTPROC lpEventProc,
   DWORD_PTR dwParam
);
```

The **InetRegisterEvent** function registers a callback function for the specified event.

Parameters

hSocket

Socket handle.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. This parameter is ignored if the socket handle specifies a server created using the **InetServerStart** function. One or more of the following values may be used:

Value	Description
INET_EVENT_ACCEPT	A network event that indicates the process has received a connection request from a client and should accept the connection using the **InetAsyncAccept** function. This event is only generated for server applications which have created an asynchronous socket using the **InetAsyncListen** function.
INET_EVENT_CONNECT	A network event that indicates the connection to the remote host has completed.
INET_EVENT_DISCONNECT	A network event that indicates the remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	A network event which indicates data is available to read. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	A network event which indicates the application can send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The application has canceled a blocking operation. This event is fired once an operation has been terminated by the **InetCancel** function, and control has been returned to the

	calling process.

lpEventProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled. This parameter cannot be NULL if the socket handle specifies a server created using the **InetServerStart** function.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetRegisterEvent** function associates a callback function with a specific event. The event handler is an **InetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the ***lpEventProc*** parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

This function can be used to change the callback function and user defined parameter for a server created using the **InetServerStart** function. However, it cannot be used with client sockets automatically created by the server interface. Those sockets are managed separately in their own thread, and individual client event notifications are not supported.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAsyncAccept, InetAsyncAcceptEx, InetAsyncListen, InetDisableEvents, InetEnableEvents, InetEventProc, InetFreezeEvents

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetReject Function

```
BOOL WINAPI InetReject(
   SOCKET hSocket
);
```

The **InetReject** function is used to reject a client connection request.

Parameters

hSocket

Handle to a listening socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReject** function rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this function will immediately return with an error indicating that the operation would cause the thread to block.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAbort, InetAccept, InetListen

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerAsyncNotify Function

```
BOOL WINAPI InetServerAsyncNotify(
    SOCKET hServer,
    HWND hWnd,
    UINT uMsg
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hServer

The socket handle.

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the unique client ID. The following notification messages are defined:

Value	Description
INET_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the **InetServerStart** function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The *lParam* message parameter will specify the handle to the server.
INET_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the

	application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the **InetServerResume** function to resume accepting client connections. The *lParam* message parameter will specify the handle to the server.
INET_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called either the **InetServerSuspend** or **InetServerSuspendEx** function. This notification message may be sent to the application multiple times over the lifetime of the server. The *lParam* message parameter will specify the handle to the server.
INET_NOTIFY_RESTART	This notification is sent when the server is restarted using the **InetServerRestart** function. Note that the server socket handle provided by the *lParam* message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The *lParam* message parameter will specify the handle to the server.
INET_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The *lParam* message parameter will specify the unique ID of the client that connected to the server.
INET_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the **InetServerStop** function. The *lParam* message parameter will specify the unique ID of the client that disconnected from the server.
INET_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in *lParam* will reference the defunct server and cannot be used with other server functions. The *lParam* message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The INET_NOTIFY_CONNECT and INET_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. If you need to obtain the handle to the client session using the ID, call the

InetGetClientHandle function. To obtain the server handle in response to the INET_NOTIFY_CONNECT message, use the **InetGetClientServerById** function. Note that at the time the application processes the INET_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

This function can only be used with a handle returned by the **InetServerStart** function and cannot be used with sockets created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientServerById, InetGetServerStatus, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerBroadcast Function

```
INT WINAPI InetServerBroadcast(
    SOCKET hServer,
    LPBYTE lpBuffer,
    INT cbBuffer
);
```

The **InetServerBroadcast** function sends data to clients that are connected to the specified server.

Parameters

hServer

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of clients that the data was sent to. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerBroadcast** function sends the contents of the buffer to all of the clients that are connected to the specified server. This function will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

This function can only be used with a socket handle created using the **InetServerStart** function and cannot be used with sockets created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetClientBroadcast, InetWrite, InetWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerLock Function

```
BOOL WINAPI InetServerLock(
    SOCKET hServer
);
```

The **InetServerLock** function locks the specified server, causing other client threads to block until it is unlocked.

Parameters

hServer

The socket handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerLock** function causes the specified server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This function should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this function, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this function will fail and return a value of zero.

Every time the **InetServerLock** function is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to **InetServerLock** must be matched by an equal number of calls to the **InetServerUnlock** function. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetLockedServer, InetServerUnlock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerRestart Function

```
SOCKET WINAPI InetServerRestart(
    SOCKET hServer
);
```

The **InetServerRestart** function restarts the server, terminating all active client sessions.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the new socket handle for the specified server. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerRestart** function will restart the specified server, terminating all active client sessions and recreating the listening socket. The socket handle that is returned by the function is the handle for the new listening socket, and the old handle value is no longer valid. If the function is unable to recreate the listening socket for any reason, the server thread is terminated.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerResume, InetServerStart, InetServerStop, InetServerSuspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerResume Function

```
BOOL WINAPI InetServerResume(
    SOCKET hServer
);
```

The **InetServerResume** function resumes accepting client connections on the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerResume** function instructs the server to resume accepting client connections. Any pending client connections that were requested while the server was suspended will be accepted.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerRestart, InetServerStart, InetServerStop, InetServerSuspend

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerStart Function

```
SOCKET WINAPI InetServerStart(
   LPCTSTR lpszLocalHost,
   UINT nLocalPort,
   UINT nBacklog,
   UINT nMaxClients,
   UINT nTimeout,
   UINT nPriority,
   DWORD dwOptions,
   INETEVENTPROC lpEventProc,
   DWORD_PTR dwEventParam,
   LPSECURITYCREDENTIALS lpCredentials
);
```

The **InetServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the socket should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

nMaxClients

The maximum number of client connections that can be established with the server. A value of zero specifies that there should not be any fixed limit on the number of active client connections. This value can be adjusted after the server has been created by calling the **InetServerThrottle** function.

nTimeout

The number of seconds the server should wait for a client to perform a network operation. If the client does not exchange any information with the server within this period of time, a timeout event will occur. The timeout value affects all clients that are connected to the server.

nPriority

An integer value which specifies the priority for the server and all client sessions. The priority for a specific client session may be modified by calling the **InetSetClientPriority** function. This parameter may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most

	applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It can be used when it is important for the client session thread to be highly responsive. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The thread will be given higher scheduling priority and will be more responsive to the remote host. It is not recommended that this priority be used on a system with a single processor.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. Enabling this option will also help applications detect the physical loss of a network connection, such as an Ethernet cable being unplugged. This option does not guarantee that persistent

	connections will be maintained over long periods of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and combines smaller packets into a single larger packet when sending data to a remote host. Specifying this option can improve the responsiveness and overall throughput of applications that implement their own buffering and exchange large amounts of information.
INET_OPTION_NOINHERIT	This option prevents the server socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the client, where the client immediately initiates the TLS handshake when it connects to the server. To implement an explicit TLS session, where the client establishes a standard, non-secure connection and then sends a command to the server to initiate a secure session, you should not use this option. Instead, use the **InetEnableSecurity** function to selectively enable TLS for the client session.

lpEventProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. This parameter cannot be NULL.

dwEventParam

A user-defined integer value that is passed to the callback function.

lpCredentials

Pointer to credentials structure SECURITYCREDENTIALS. This may be NULL, unless the ***dwOptions*** parameter includes INET_OPTION_SECURE. When a secure session is specified, the fields ***dwSize***, ***lpszCertStore***, and ***lpszCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

In most cases, the ***lpszLocalHost*** parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

When the event handler callback function is invoked by the server, it normally executes in the context of the worker thread that manages that client session. This means that even if you do not explicitly create any threads in your application, you must design your program to be thread-safe, with synchronized access to global objects and data. If your application has a user interface, only the main UI thread should attempt to modify controls. If you attempt to modify a control from a worker thread, such as adding a row to a listbox control, it can result the application becoming deadlocked. This means that you should not attempt to directly update the UI from within the event handler function. To enable asynchronous server notifications for a GUI application, use the **InetServerAsyncNotify** function.

The socket handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **InetServerStop** function.

Example

```
#define SERVER_PORT       7000
#define SERVER_CLIENTS    100

SOCKET hServer = INVALID_SOCKET;

// Accept connections from clients that connection on port 7000 with a default
// backlog of 5 connections and a maximum of 100 client connections.

hServer = InetServerStart(NULL,
                          SERVER_PORT,
                          INET_BACKLOG,
                          SERVER_CLIENTS,
                          INET_TIMEOUT,
                          INET_PRIORITY_NORMAL,
                          INET_OPTION_REUSEADDRESS,
                          MyEventHandler,
                          0,
                          NULL);

if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];
```

```
    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetServerData, InetGetServerPriority, InetGetServerStatus, InetServerLock, InetServerRestart, InetServerResume, InetServerStop, InetServerSuspend, InetServerThrottle, InetServerUnlock, InetSetServerData, InetSetServerPriority InetValidateCertificate

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerStop Function

```
BOOL WINAPI InetServerStop(
    SOCKET hServer
);
```

The **InetServerStop** function signals the server to stop listening for connections and terminates all client sessions.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerStop** function signals the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The socket handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

If this function is called when there is one or more clients connected to the server, it will signal each client thread to terminate and then wait for the server thread to terminate. As the client sessions are terminated, the event handler will not be invoked. If you wish to ensure that all clients are disconnected normally before stopping the server, call the **InetServerSuspendEx** function with the INET_SUSPEND_DISCONNECT option and then stop the server after the last client has disconnected.

Because the **InetServerStop** function waits for the server thread to terminate, this function may cause your application to block. If this is not desirable, use the **InetServerStopEx** function which can perform the shutdown sequence asynchronously.

After the server thread has been terminated, the listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, the option INET_OPTION_REUSEADDRESS must be specified when calling the **InetServerStart** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerRestart, InetServerStart, InetServerStopEx, InetServerSuspendEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerStopEx Function

```
BOOL WINAPI InetServerStopEx(
   SOCKET hServer,
   DWORD dwMilliseconds
);
```

The **InetServerStopEx** function signals the server to stop listening for connections and terminates all client sessions.

Parameters

hServer

Handle to the server socket.

dwMilliseconds

An unsigned integer value that specifies the number of milliseconds to wait for all active clients to disconnect and the server thread to terminate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerStopEx** function signals the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The socket handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

Unlike the **InetServerStop** function, which waits for fixed period of time for the server thread to terminate, the **InetServerStopEx** allows the caller to determine how much time should be spent waiting for the clients to disconnect and the server thread to terminate. If the ***dwMilliseconds*** parameter has a value of INFINITE, the function will wait for an indefinite period of time until all clients have disconnected, the listening socket closed and the server thread has terminated. If the ***dwMilliseconds*** parameter has a value of zero, the function does not wait for the server to shutdown. Instead, it returns immediately and the shutdown process continues in the background.

If your application specifies a value of zero for the ***dwMilliseconds*** parameter, the event handler will be invoked with the INET_EVENT_DISCONNECT event as each client disconnects from the server during the shutdown process. If you depend on this event to perform some cleanup on a per-client basis, you must ensure that the application does not exit until the server thread has terminated. To perform a graceful shutdown of the server, it is recommended that you use the **InetServerSuspendEx** function and specify the INET_SUSPEND_DISCONNECT option. After all clients have disconnected, call the **InetServerStop** function to terminate the server thread.

After the server thread has been terminated, the listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, the option INET_OPTION_REUSEADDRESS must be specified when calling the **InetServerStart** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerRestart, InetServerStart, InetServerStop, InetServerSuspendEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerSuspend Function

```
BOOL WINAPI InetServerSuspend(
    SOCKET hServer
);
```

Suspend accepting client connections on the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be queued up to the maximum backlog value specified when the server was started. To resume accepting client connections, call the **InetServerResume** function.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **InetServerThrottle** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerRestart, InetServerResume, InetServerStart, InetServerStop, InetServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerSuspendEx Function

```
BOOL WINAPI InetServerSuspendEx(
    SOCKET hServer,
    DWORD dwOptions
);
```

Suspend accepting client connections and optionally reject or disconnect clients.

Parameters

hServer

Handle to the server socket.

dwOptions

An unsigned integer that specifies one or more options.

Value	Description
INET_SUSPEND_DEFAULT	Specifies that the server should suspend accepting new connections. New incoming client connections will be queued and clients that have already established a connection to the server will remain connected.
INET_SUSPEND_REJECT	Specifies that the server should suspend accepting new connections and reject any new client connections. Clients that have already established a connection to the server will remain connected.
INET_SUSPEND_DISCONNECT	Specifies that the server should suspend accepting new connections and disconnect all active clients that are currently connected to the server. If the INET_SUSPEND_REJECT option is also specified, new client connections will be rejected by the server.
INET_SUSPEND_WAIT	Specifies that the function should wait for the clients to disconnect from the server rather than return immediately to the caller. This option is only meaningful when used in conjunction with INET_SUSPEND_DISCONNECT.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerSuspendEx** function instructs the server to suspend accepting new client connections. Additional options can be used to control how the server suspends connections. If you plan on suspending the server for any period of time greater than a few seconds, it is recommended that you specify the INET_SUSPEND_REJECT option so that clients are not forced to wait.

If the INET_SUSPEND_DISCONNECT option is specified, the server will signal each client to disconnect and will stop accepting new connections. The event handler will be invoked for each client that disconnects. If the INET_SUSPEND_WAIT option is also specified, the function will wait until the last client has disconnected from the server before returning to the caller. If there are a large number of

clients connected to the server, this process may cause the application to block for an extended period of time and appear to be non-responsive to the user. For this reason, you should not specify the INET_SUSPEND_WAIT option if the function is being called from the application's main UI thread.

To perform a graceful shutdown of the server, it is recommended that you call **InetServerSuspendEx** with the INET_SUSPEND_REJECT and INET_SUSPEND_DISCONNECT options. This will allow each client to disconnect from the server and the server will reject any new incoming connections. After the last client has disconnected from the server, call the **InetServerStop** function to complete the shutdown process. The **InetGetClientThreads** function can be used to determine if all client session threads have terminated.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerRestart, InetServerResume, InetServerStart, InetServerStop, InetServerThrottle

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerThrottle Function

```
BOOL WINAPI InetServerThrottle(
   SOCKET hServer,
   UINT nMaxClients,
   UINT nMaxClientsPerAddress,
   DWORD dwConnectionRate
);
```

The **InetServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server socket.

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is no limit on the number of client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **InetServerStart** function is called with the maximum number of clients set to 100, and then **InetServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept

incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetServerStatus, InetServerLock, InetServerRestart, InetServerResume, InetServerStart, InetServerSuspend, InetServerUnlock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetServerUnlock Function

```
BOOL WINAPI InetServerUnlock(
    SOCKET hServer
);
```

The **InetServerUnlock** function unlock the specified server, allowing other client threads to resume execution.

Parameters

hServer

The socket handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerUnlock** function releases the lock on the specified server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **InetServerLock** function is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to **InetServerLock** must be matched by an equal number of calls to the **InetServerUnlock** function. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this function, and should never assume that the lock has been released. If a potential deadlock situation is detected, this function will fail and return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetLockedServer, InetGetServerStatus, InetServerLock

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetClientData Function

```
BOOL WINAPI InetSetClientData(
    SOCKET hClient,
    LPVOID lpvData
);
```

The **InetSetClientData** function sets the application defined data associated with the specified client session.

Parameters

hClient

The client socket handle.

lpvData

Pointer to the application defined data associated with the specified client session.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be modified. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetClientData** function is used to associate application defined data with a specific client session. This is typically used to associate a pointer to a data structure or a class instance with the client socket. The data can be retrieved using the **InetGetClientData** function.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling function exits. If you do this, the pointer will no longer be valid after the function exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a function such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization functions when dereferencing and modifying that structure.

This function can only be used with client socket handles created using the SocketWrench server interface. It cannot be used with socket handles created using the **InetConnect** or **InetAccept** functions.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));
UINT *pnValue2 = NULL;

*pnValue1 = 1234;

if (InetSetClientData(hClient, pnValue1) == FALSE)
{
    // Unable to associate the data with this session
    return;
}

if (InetGetClientData(hClient, &pnValue2) == FALSE)
{
    // Unable to retrieve the data associated with this session
    return;
```

```
}

// *pnValue2 == 1234
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientData, InetGetServerData, InetSetServerData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetClientMoniker Function

```
INT WINAPI InetSetClientMoniker(
   SOCKET hSocket,
   LPCTSTR lpszMoniker
);
```

The **InetSetClientMoniker** function associates a unique string moniker with the specified client session.

Parameters

hSocket

Handle to the client socket.

lpszMoniker

Pointer to a string which specifies the moniker for the specified client socket. If this parameter is NULL or specifies an empty string, a moniker will no longer be associated with the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. The **InetGetClientMoniker** function will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **InetFindClientMoniker** function.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetFindClientMoniker, InetGetClientHandle, InetGetClientId, InetGetClientMoniker

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetClientPriority Function

```
INT WINAPI InetSetClientPriority(
    SOCKET hClient,
    INT nPriority
);
```

The **InetSetClientPriority** function sets the current priority for the specified client session.

Parameters

hClient

Handle to the client session.

nPriority

An integer value which specifies the new priority for the client session. It may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority for the specified client session. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetClientPriority** function can be used to change the current priority assigned to the specified client session. The client priority is inherited from the priority specified when the server is started using the **InetServerStart** function.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientPriority, InetGetServerPriority, InetServerStart, InetSetServerPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetHostFile Function

```
INT WINAPI InetSetHostFile(
   LPCTSTR lpszFileName
);
```

The **InetSetHostFile** function specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the function succeeds, the return value is the number of entries in the host file. A return value of INET_ERROR indicates failure. To get extended error information, call **InetGetLastError**.

Remarks

This function loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this function again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **InetGetHostFile** function. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **InetGetDefaultHostFile** function. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetDefaultHostFile, InetGetHostAddress, InetGetHostFile, InetGetHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetLastError Function

```
VOID WINAPI InetSetLastError(
   DWORD dwErrorCode
);
```

The **InetSetLastError** function sets the last-error code for the caller. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last-error code for the caller.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those functions which call **InetSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the InetGetLastError function. The use of **InetGetLastError** is optional; an application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetErrorString, InetGetLastError

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetOption Function

```
INT WINAPI InetSetOption(
   SOCKET hSocket,
   DWORD dwOption,
   BOOL bEnabled
);
```

The **InetSetOption** function is used to enable or disable a specific socket option.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

bEnabled

A boolean flag. If the flag is set to a non-zero value, the option is enabled. Otherwise the socket option is disabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

If if the INET_OPTION_KEEPALIVE option is enabled, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetAsyncConnectEx, InetConnectEx, InetGetOption

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetServerData Function

```
BOOL WINAPI InetSetServerData(
    SOCKET hServer,
    LPVOID lpvData
);
```

The **InetSetServerData** function sets the application defined data associated with the specified server.

Parameters

hServer

The socket handle.

lpvData

Pointer to the application defined data associated with the specified server.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the server could not be modified. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerData** function is used to associate application defined data with a specific server. The data can be retrieved using the **InetGetServerData** function.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling function exits. If you do this, the pointer will no longer be valid after the function exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a function such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization functions when dereferencing and modifying that structure.

This function can only be used with server socket handles created using the **InetServerStart** function. It cannot be used with socket handles created using the **InetListen** or **InetListenEx** functions.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));
UINT *pnValue2 = NULL;

*pnValue1 = 1234;

if (InetSetServerData(hServer, pnValue1) == FALSE)
{
    // Unable to associate the data with this server
    return;
}

if (InetGetServerData(hServer, &pnValue2) == FALSE)
{
    // Unable to retrieve the data associated with this server
    return;
}

// *pnValue2 == 1234
```

```
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientData, InetGetServerData, InetSetClientData

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetServerPriority Function

```
INT WINAPI InetSetServerPriority(
    SOCKET hServer,
    INT nPriority
);
```

The **InetSetServerPriority** function sets the current priority for the specified server.

Parameters

hServer

Handle to the server socket.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function. To modify the priority for an active client session, use the **InetSetClientPriority** function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetClientPriority, InetGetServerPriority, InetServerStart, InetSetClientPriority

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetServerStackSize Function

```
BOOL WINAPI InetSetServerStackSize(
    SOCKET hServer,
    DWORD dwStackSize
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetMemoryUsage, InetGetServerStackSize, InetServerStart

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetSetTimeout Function

```
INT WINAPI InetSetTimeout(
   SOCKET hSocket,
   UINT nTimeout
);
```

The **InetSetTimeout** function sets the interval that is used when waiting for a blocking operation to complete.

Parameters

hSocket

Handle to the socket.

nTimeout

Duration of timeout interval, in seconds. If a value over 1000 is specified, it is assumed that milliseconds are intended by the user, and the value actually used will be adjusted accordingly.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetTimeout, InetConnect, InetAccept, InetIsReadable, InetIsWritable

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetShutdown Function

```
INT WINAPI InetShutdown(
   SOCKET hSocket,
   DWORD dwOption
);
```

The **InetShutdown** function is used to disable reception or transmission of data, or both.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the shutdown options. These options cannot be combined. The following values are recognized:

Value	Description
INET_SHUTDOWN_READ	Disable reception of data.
INET_SHUTDOWN_WRITE	Disable transmission of data.
INET_SHUTDOWN_BOTH	Disable both reception and transmission of data.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function is rarely needed. It is provided as an interface to the Windows Sockets **shutdown** function.

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **InetShutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetDisconnect

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetStoreStream Function

```
BOOL WINAPI InetStoreStream(
   SOCKET hSocket,
   LPCTSTR lpszFileName,
   DWORD dwLength,
   LPDWORD lpdwCopied
   DWORD dwOffset,
   DWORD dwOptions
);
```

The **InetStoreStream** function reads the socket data stream and stores the contents in the specified file.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the function will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the function returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the function will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Value	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream. If the *dwOffset* parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetStoreStream** function enables an application to read an arbitrarily large stream of data and store it in a file. This function is essentially a simplified version of the **InetReadStream** function, designed specifically to be used with files rather than memory buffers or handles.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetStoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **InetStoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
DWORD dwCopied;
BOOL bResult;

bResult = InetStoreStream(hSocket,
                          lpszFileName,                          0,
                          &dwCopied,
                          0,
                          INET_STREAM_CONVERT);

if (bResult && dwCopied > 0)
{
    // The data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetGetStreamInfo, InetRead, InetReadLine, InetReadStream, InetWrite, InetWriteLine, InetWriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetUninitialize Function

```
VOID WINAPI InetUninitialize();
```

The **InetUninitialize** function terminates the use of the library.

Parameters

None.

Return Value

None.

Remarks

An application is required to perform a successful **InetInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **InetUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **InetUninitialize** for every successful call to **InetInitialize** made by a process. In a multithreaded environment, operations for all threads are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetDisconnect, InetInitialize

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetValidateCertificate Function

```
BOOL WINAPI InetValidateCertificate(
   LPCTSTR lpszCertStore,
   LPCTSTR lpszCertPassword,
   LPCTSTR lpszCertName
);
```

The **InetValidateCertificate** function determines if the specified security certificate is installed on the local system.

Parameters

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the personal certificate store will be used as the default. This parameter may also specify the name of a certificate file in PKCS #12 (PFX) format.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertPassword

A null-terminated string which specifies the password associated with a certificate file. This parameter is only used if the ***lpszCertStore*** parameter specifies a certificate file, otherwise it is ignored. If the certificate file is not protected with a password, this parameter should be a NULL pointer or empty string.

lpszCertName

A null-terminated string which specifies the name of the certificate to validate. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This

comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

If you are checking the validity of a certificate installed in the local certificate store, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user.

It is possible to validate a certificate file rather than one stored in the local certificate store. The *lpszCertStore* member should specify the name of a file in Private Information Exchange (PFX) format, also known as PKCS #12.These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate file was created, it must be provided in with the *lpszCertPassword* parameter or this function will be unable to access the certificate.

This function can only validate certificate files in PFX format and cannot be used to validate a certificate file in another format, such as PEM (base64 encoded) or DER (binary).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetCreateSecurityCredentials, InetDeleteSecurityCredentials

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetValidateHostName Function

```
BOOL WINAPI InetValidateHostName(
   LPCTSTR lpszHostName,
   LPTSTR lpszAddress,
   INT nMaxLength
);
```

The **InetValidateHostName** function determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The function will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the ***nMaxLength*** parameter is zero, it will be ignored and the IP address will not be returned.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the ***lpszAddress*** string buffer. The buffer must be large enough to store the complete address. Because this function can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the ***lpszAddress*** parameter will be ignored.

Return Value

If the function succeeds, the host name is valid and the return value will be non-zero. If the function fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetValidateHostName** function provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. It is similar to calling the **InetNormalizeHostName** function to obtain the canonical form of the host name, calling **InetGetAddress** to obtain the IP address and then calling **InetFormatAddress** to return the string representation of the host's IP address.

If the Unicode version of this function is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

The ***lpszHostName*** parameter can only specify a host name or IP address and cannot be a URL. If you want your application to support providing a URL in addition to a host name, use the **InetGetUrlHostName** function to extract the host name from the URL. You can then provide the host name to this function to obtain its IP address.

If the ***lpszHostName*** parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if

the IP address has a corresponding host name, use the **InetGetHostName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetFormatAddress, InetGetAddress, InetGetHostAddress, InetGetHostName, InetGetUrlHostName, InetNormalizeHostName

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetWrite Function

```
INT WINAPI InetWrite(
   SOCKET hSocket,
   LPBYTE lpBuffer,
   INT cbBuffer
);
```

The **InetWrite** function sends the specified number of bytes to the remote host.

Parameters

hSocket

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the application to send the remaining data at some later point. For non-blocking connections, the program must wait for the INET_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetFlush, InetRead, InetReadEx, InetWriteEx

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetWriteEx Function

```
INT WINAPI InetWriteEx(
   SOCKET hSocket,
   LPVOID lpvBuffer,
   INT cbBuffer,
   DWORD dwReserved,
   LPINTERNET_ADDRESS lpRemoteAddress,
   UINT nRemotePort
);
```

The **InetWriteEx** function sends the specified number of bytes to the remote host.

Parameters

hSocket

The socket handle.

lpvBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer.

dwReserved

Reserved parameter. This value must always be zero.

lpRemoteAddress

Pointer to an INTERNET_ADDRESS structure that specifies the address of the remote host that is to receive the data being written. For TCP stream sockets, this parameter must always be NULL or specify the same address that was used to establish the connection. For UDP datagram sockets, this may specify any valid IP address.

nRemotePort

The port number of the remote host that is to receive the data being written. For TCP stream sockets, this value must always be zero, or specify the same port number that was used to establish the connection. For UDP datagram sockets, this may specify any valid port number.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the ***cbBuffer*** parameter. In this case, the data has been partially written and it is the responsibility of the application to send the remaining data at some later point. For non-blocking connections, the program must wait for the INET_EVENT_WRITE asynchronous notification message before it resumes sending data.

This function extends the **InetWrite** function to additional information about the destination IP address and port number for the data being written. For a client TCP connection, the IP address and remote port must be the same values that were used to establish the connection. When writing on a UDP socket, this is the IP address and remote port of the peer that will receive the datagram. This information can be used in conjunction with the **InetReadEx** function to send a datagram back to that host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetFlush, InetRead, InetReadEx, InetWrite, INTERNET_ADDRESS

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetWriteLine Function

```
BOOL WINAPI InetWriteLine(
   SOCKET hSocket,
   LPCTSTR lpszBuffer,
   LPINT lpnLength
);
```

The **InetWriteLine** function sends a line of text to the remote host, terminated by a carriage-return and linefeed.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetWriteLine** function writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **InetWrite** function which writes arbitrary bytes of data to the socket, this function is specifically designed to write a single line of text data from a string.

If the ***lpszBuffer*** string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the ***lpnLength*** parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **InetWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this function is called with

asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **InetWriteLine** is blocked sending data to the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **InetWrite** and **InetWriteLine** function calls can be safely intermixed.

Unlike the **InetWrite** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the ***lpnLength*** argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib
Unicode: Implemented as Unicode and ANSI versions

See Also

InetIsWritable, InetRead, InetReadLine, InetWrite

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InetWriteStream Function

```
BOOL WINAPI InetWriteStream(
   SOCKET hSocket,
   LPVOID lpvBuffer,
   LPDWORD lpdwLength,
   DWORD dwOptions
);
```

The **InetWriteStream** function writes data from the stream buffer to the specified socket.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the ***dwOptions*** parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the function returns. This argument should should always point to an initialized value. If the ***lpvBuffer*** argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the ***lpvBuffer*** parameter. If the argument specifies a a global memory handle, then the function will write the data referenced by that handle; otherwise, the function will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The ***lpvBuffer*** argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the ***lpdwLength*** argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The ***lpvBuffer*** argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream

	buffer type is used, the *lpdwLength* argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The *lpvBuffer* argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The *lpvBuffer* argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the *hSocket* parameter. The socket handle passed to this function must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either attach the socket using the InetAttachSocket function or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetWriteStream** function enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **InetWrite** function, which may not write all of the data in a single function call, **InetWriteStream** will only return when all of the data has been written or an error occurs.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetWriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for some data to have been written even if the function returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the function is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **InetWriteStream** can potentially cause the application to block for long periods of time as the data stream is being written, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
HANDLE hFile;
```

```
DWORD dwLength;

hFile = CreateFile(lpszFileName,
                   GENERIC_READ,
                   FILE_SHARE_READ,
                   NULL,
                   OPEN_EXISTING,
                   FILE_FLAG_SEQUENTIAL_SCAN,
                   NULL);

if (hFile == INVALID_HANDLE_VALUE)
    return;

dwLength = GetFileSize(hFile, NULL);

if (dwLength > 0)
{
    BOOL bResult = InetWriteStream(
                     hSocket,
                     hFile,
                     &dwLength,
                     INET_STREAM_HANDLE);
}

CloseHandle(hFile);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h
Import Library: cswskv12.lib

See Also

InetGetStreamInfo, InetRead, InetReadLine, InetReadStream, InetStoreStream, InetWrite, InetWriteLine

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS
- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO

The INETSTREAMINFO structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD   dwStreamThread;
    DWORD   dwStreamSize;
    DWORD   dwStreamCopied;
    DWORD   dwStreamMode;
    DWORD   dwStreamError;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Value	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

See Also

InetReadStream, InetStoreStream, InetWriteStream

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the ***ipNumber*** array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the ***ipNumber*** array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the ***ipNumber*** array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The INTERNET_ADDRESS structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the ***ipNumber*** array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the ***ipFamily*** member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cswsock12.h

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpszCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpszPassword*** member or the library will be unable to access the certificate.

Note that the ***lpszUserName*** and ***lpszPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SECURITYINFO

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
   DWORD        dwSize;
   DWORD        dwProtocol;
   DWORD        dwCipher;
   DWORD        dwCipherStrength;
   DWORD        dwHash;
   DWORD        dwHashStrength;
   DWORD        dwKeyExchange;
   DWORD        dwCertStatus;
   SYSTEMTIME   stCertIssued;
   SYSTEMTIME   stCertExpires;
   LPCTSTR      lpszCertIssuer;
   LPCTSTR      lpszCertSubject;
   LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpszCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpszFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpszCertIssuer*** and ***lpszCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1
Minimum Server Platform: Windows Server 2008 R2 Service Pack 1
Header File: cstools12.h
Unicode: Implemented as Unicode and ANSI versions

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench Library Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of function parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to function
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to function
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to function
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to function
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified

0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence

0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This function has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session

0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections

Copyright © 2026 Catalyst Development Corporation. All rights reserved.